JØRN LORENTZEN OG HENRIK NELLAGER

Maskinkode med ZX81

Jørn Lorentzen og
Henrik Nellager

Maskinkode med ZX 81

Borgen

Copyright © 1985 Jørn Lorentzen og
Henrik Nellager
Omslag: Patrick Scott-Fordsman
Trykt hos Narayana Press
ISBN 87-418-7245-2

Alle rettigheder forbeholdes, herunder retten til erhvervsmæssig udnyttelse af bogens programmer.

Indhold

		Side
Forord		9
Introduktion		11-12
Kapitel 1	Opbygning af ZX 131 og dens RAM Vi forklarer, hvad ZX 81 består af, og desuden fortæller vi om brugerlagerets (RAM's) struktur.	13-17
Kapitel 2	Talsystemer Vi lærer de heksadecimale og binære talsystemer at kende, da disse er grundlæggende for maskinkodeprogrammering.	18-20
Kapitel 3	LD-instruktionen Vi lærer at tildele registre værdi samt at PEEK'e og POKE'e i maskinkode. Dette er den mest brugte instruktion. Endvidere kigger vi på flagregisteret.	21-33
Kapitel 4	Simpel aritmetik Vi fortæller om instruktionerne INC, DEC, ADD, ADC, SUB og SBC, med hvilke vi kan addere og subtrahere.	34-41
Kapitel 5	Specielle registre Vi kigger nærmere på de specielle registre i Z80A-processoren, og vi lærer herved at udtrykke negative tal.	42-48
Kapitel 6	De logiske instruktioner Vi omtaler AND, OR og XOR, der sammenligner enkelte bits med hinanden.	49-55
Kapitel 7	Hop-instruktioner Vi lærer at hoppe frem og tilbage i programmer med instruktionerne JP, JR og DJNZ.	56-62
Kapitel 8	Sammenligningsinstruktioner Vi lærer at sammenligne registre med hinanden ved hjælp af CP-instruktionen, og vi stifter bekendtskab med en ombytningsinstruktion.	63-68

Kapitel 9 Simple instruktioner 69-71
Vi gennemgår en række meget simple instruktioner. Det er CPL, NEG, CCF, SCF, NOP, SET, RES og BIT.
Kapitel 10 Maskinstakken 72-79
Vi lærer at gemme tal et sted i RAM med instruktionen PUSH, og vi får dem tilbage med POP.
Kapitel 11 Subrutiner 80-88
Vi kalder subrutiner med CALL og returnerer med RET; ikke blot kald til rutiner i RAM, men også i ROM.
Kapitel 12 Specielle subrutine-kald 89-94
Vi omtaler RST-instruktionerne, som kalder bestemte rutiner i ROM.
Kapitel 13 Blokinstruktioner I 95-101
Vi lærer at flytte hele blokke af data med instruktionerne LDI, LDD, LDIR og LDDR.
Kapitel 14 Blokinstruktioner II 102-104
Vi sammenligner blokke af data ved hjælp af instruktionerne CPI, CPD, CPIR og CPDR.
Kapitel 15 Rotationsinstruktioner 105-116
Vi lærer at flytte rundt på de enkelte bits i registrene med RLC, RRC, RL, RR, SLA, SRA, SRL, RRD og RLD.
Kapitel 16 De alternative registre 117-122
Vi lærer de passive registre at kende, og vi ombytter med de nuværende ved hjælp af EXX. Endvidere ser vi på BCD-aritmetik med DAA-instruktionen.
Kapitel 17 De svære instruktioner 123-126
Vi kigger på de svære instruktioner i Z80A-instruktionssættet. Det er IN, OUT, EI, DI, HALT, RETN, RETI, IM samt blok-instruktioner til IN og OUT.
Kapitel 18 Oversigt 127-130
Vi gennemgår samtlige instruktioner.
Kapitel 19 Tastaturet 131-135
Vi lærer at aflæse tastaturet.

Kapitel 20 ROM-rutiner 136-147
Vi fortæller, hvor forskellige rutiner er placeret i ROM, og vi giver en opskrift på, hvordan du kan lave en disassembler.
Kapitel 21 Gode råd 148-150
Vi fortæller, hvor det er bedst at lagre maskinkoderutiner, og vi giver gode råd om programmering.
Kapitel 22 Beregnerstakken. 151-164
Vi lærer at foretage indviklede matematiske beregninger med RST 28h-instruktionen.
Stikordsregister 237-240

Programeksempler:
Program 1 Typewriter 165-166
Program 2 2. grads ligning 167-174
Program 3 Grafer 175-177
Program 4 Moon Lander 178-185
Program 5 Space Invaders 186-213
Program 6 Musik på ZX 81 214-216

Appendikser:
Appendiks A Fra heks-koder til assembler 217-228
Appendiks B Systemvariable 229-230
Appendiks C Konvertering mellem talsystemer 231-232
Appendiks D Flagsætning 233-236

Større programmer:
Kapitel 3 side 15 Heks-loader (BASIC) 21
Kapitel 7 side 52 Invertering af skærmbillede 58
Kapitel 8 side 57 Renumber 68
Kapitel 13 side 87 Scroll 98
Kapitel 13 side 91 Antiscroll 101
Kapitel 15 side 104 Forstørrelse af tegn 115
Kapitel 16 side 110 Omskrivning fra decimale til heksadecimale tal 121

Forord

Maskinkode er ukendt for mange. Måske også for dig. Med denne bog får du indblik i dette avancerede programmeringssprog. Det er vort formål at få udbredt kendskabet til sproget. Vi har ved udarbejdelsen lagt vægt på, at du lærer selv at arbejde med maskinkode, så du ved bogens slutning er i stand til at lave komplicerede programmer, der udvider datamatens kapacitet.

Vi har baseret bogen på ZX 81, men den kan også benyttes til Lambda 8300. Visse af de programeksempler, som vi giver undervejs, virker kun på ZX 81. Selve maskinkodeinstruktionerne er de samme som for andre datamater med CPU'en Z80 eller Z80A, blot kan programmerne ikke bruges uden tilpasning til den enkelte datamat.

Du kan benytte bogen uden specielle forudsætninger. Det vil være en fordel, hvis du har indsigt i BASIC-programmering, men det er absolut ingen nødvendighed! Vi går langsomt frem, så du lærer maskinkode grundigt at kende helt fra bunden.

Vi ønsker dig rigtig god fornøjelse!

Jørn Lorentzen & Henrik Nellager

Introduktion

Hvorfor nu lære maskinkode? Hvad er maskinkode? Hvad er fordelene? Har det mange ulemper? Du kan sikkert nævne mange flere spørgsmål, som trænger sig på. I denne introduktion vil vi prøve at beskrive maskinkode på godt og ondt.

Du har sikkert allerede nu brugt utallige timer på din ZX 81. Du har programmeret på den i BASIC, og nu ønsker du at komme videre. Det kan du også komme. BASIC er et højniveausprog. ZX 81's elektroniske enheder forstår ikke BASIC, de forstår kun at skelne imellem to ting: lav og høj spænding. Blandt mange komponenter indeholder ZX 81 en BASIC-fortolker, der oversætter din BASIC til et sprog, som de elektroniske dele forstår. Dette sprog hedder maskinkode eller maskinsprog.

Maskinkode er egentlig blot en række tal. Man har dog nu udbygget dette lavniveausprog, så vi mennesker har fået nogle ord at programmere med. I BASIC kaldes det kommandoer og funktioner, i maskinkode instruktioner. Disse ord, der gør det nemmere for os, kaldes mnemonics, og man kalder det at programmere i assembler.

Hvorfor nu lære det? Jo, ved at programmere i maskinkode undgår man BASIC-fortolkeren, og herved spares en masse tid. Maskinkode er så ufattelig hurtigt, at de første timer, du arbejder med det, forstår du nok ikke, at det kan være så effektivt. Man kan se det som en fordel, at man arbejder direkte med de elektroniske enheder, men sproget er sværere end BASIC. Man må holde sig til nogle meget simple instruktioner, men efter træning og indlæring – sidstnævnte proces hjælper vi dig med – bliver det nemmere. Du skal ikke fortryde, at du har købt denne bog. Den vil give dig indblik i din datamats arbejdsrytme.

Der er én ulempe ved maskinkode, som man skal vænne sig til. Hvis man laver fejl i BASIC, vil programmet sædvanligvis stoppe med fejlrapporter, men det kan også give forkerte resultater ved kørsel. I begge tilfælde vil man normalt hurtigt finde

fejlen, og der er ingen ulykke sket. I maskinkode vil en fejl formentlig have samme virkning som at slukke datamaten. Når datamaten begynder at opføre sig usædvanligt, opstår et såkaldt crash, og man er nødt til at slukke datamaten. Man må begynde helt forfra! – uden at få at vide, hvor fejlen er, og den kan være svær at spore.

På Lambda 8300 vil man i visse tilfælde kunne stoppe et fejlkørende maskinkodeprogram ved at trykke på RESET-knappen. Lambda-ejere skal ikke fortvivle, når vi i bogen hele tiden taler om ZX 81. Er intet andet nævnt, vil der ikke være forskel mellem ZX 81 og Lambda. Når der er forskel, er koden til Lambda anført i parentes nedenunder.

Har du ingen RAM-udvidelse, men kun 1 eller 2 K RAM, kan du få svært ved at bruge eksemplerne, og vi anbefaler dig, for at du kan få det fulde udbytte af bogen, at du anskaffer dig et sådant udvidelsesmodul på 16 K eller mere. Der er ikke albuerum nok uden denne udvidelse, og da slet ikke når du selv skal programmere.

Glæden ved at arbejde med maskinkode kan vi nok bedst beskrive med et citat fra bogen »Maskinen« af Tracy Kidder. Her fortæller en medarbejder i et datafirma om sit møde med assemblerprogrammering:
»Det var dejligt at lære det. Jeg kunne springe mellemmændene over og tale direkte med maskinen. Det var en enorm oplevelse for mig at lære dette præstesprog. Jeg kunne tale med Gud«.

Kapitel 1

ZX 81 består af fire hovedenheder, der alle er nødvendige for at få en mikrodatamat til at fungere. "Hjernen" bag datamaten er den såkaldte CPU (Central Processing Unit). Den styrer lagrene, de indkørte data, tasterne, billedet og meget mere. CPU'en i ZX 81 er den meget benyttede Z80A fra Zilog, den processor, vi i løbet af denne bog skal lære at programmere direkte.

Den anden enhed, som er nødvendig, er en såkaldt ROM (Read Only Memory). Denne kreds er fastlagt af producenten og kan derfor ikke slettes. ROM indeholder en BASIC-fortolker, en kalkulator og en tegngenerator. BASIC-fortolkeren oversætter programmørens BASIC til maskinkode, som CPU'en kan forstå. Det er denne fortolker, som vi vil lære at gå udenom, så vi kommer i »direkte kontakt« med CPU'en. Kalkulatoren foretager beregninger, mens tegngeneratoren danner de tegn, der skal vises på skærmen.

Den tredje del er et arbejdslager. Vi behøver et sted til aflevering af vores programmer, hvor de så kan gemmes. Til dette formål har vi RAM (Random Access Memory). Den kan ændres efter vort ønske, og det er denne del, som du måske har udvidet med en 16 K, 32 K eller 64 K RAM. Den gemmer ikke blot programmer, men også data om skærmbilledet samt andre oplysninger, som ZX 81 behøver til internt brug.

Den fjerde hovedenhed er den såkaldte ULA, og det er denne kreds, der dirigerer trafikken på ZX 81's ufremkommelige veje. I ZX 81 hedder denne chip SCL (Sinclair Computer Logic).

Vi har nu belyst de fire vigtigste integrerede kredse i ZX 81's indre. Den vigtigste, set med programmørens øjne, er naturligvis RAM, for denne er let tilgængelig og kan ændres efter behov.

Arbejdslageret består af en række bytes, som hver består af 8 bits. Hver byte indeholder et tal mellem 0 og 255; men en nær-

mere forklaring vender vi tilbage til i næste kapitel. Hver byte har et nummer. ZX 81 har fra fødslen 1 K RAM, dvs. 1024 bytes, men øges lageret til 16 K, har man 16.384 bytes til rådighed. Disse bytes ligger på en lang række i ZX 81 – man siger, at de ligger på hver sin adresse. Den første byte ligger på adresse 16384, den næste på 16385 osv. Har man 16 K RAM, vil den sidste adresse være 32767.

Adresserne fra 0 til 8191 optages af ROM (ZX 81 har en 8 K ROM), og adresserne fra 8192 og 16383 er gemt til en evt. udvidelse af denne. Indtil videre ligger der blot en kopi af ROM deri.

RAM består af forskellige områder. Det første område indeholder altid 125 bytes, og det er de såkaldte systemvariable, der ligger her. Systemvariable er et antal variable, som ZX 81 bruger til at checke forskellige ting, såsom skærmbilledet, placeringen af de forskellige enheder i RAM og meget andet.

16384	Systemvariable	Beregnerstak
16509	Programfil	Reserve
	Udskriftfil	Maskinstak
	Variabelfil	GOSUB-stak
	E-line	RAMTOP
	Linje- + syntakskontrol	

Visse af disse systemvariable kan være nyttige for programmøren. Her vil vi kun trække de vigtigste frem, men flere vil blive omtalt i de kapitler, hvor de bliver benyttet. En fuldstændig liste for ZX 81 findes i appendiks B.

En af de vigtige er den såkaldte RAMTOP. RAMTOP er den første adresse, der ikke findes i BASIC-systemet. Man kan sænke RAMTOP, f.eks. 100 bytes, så der bliver 100 bytes fri ovenover, men dette vender vi tilbage til under placeringen af maskinkode i kapitel 21. Den systemvariabel, der fortæller ZX 81, hvor RAMTOP er, fylder to bytes og ligger på adresserne 16388 og 16389. For at finde ud af, hvad der er placeret i en given byte, bruger vi BASIC-funktionen PEEK. Tilsvarende kan man lægge en ny værdi ind på en adresse med kommandoen POKE. Adressen fås ved at skrive:


```
PRINT PEEK 16388+256*PEEK 16389
```

Som det ses, skal indholdet af byte 16389 multipliceres med 256. Det skyldes, at når et tal over 255 skal placeres, behøves to bytes, da en byte kun kan indeholde tal mellem 0 og 255. Den ene byte tæller så antallet af 256'ere, den anden 1'ere. Tallet 1000 ville altså kunne skrives som 3 x 256 + 232, og det vil sige, at den byte, der rummer det mest betydende indhold, den såkaldte MSB (Most Significant Byte = mest vigtige byte), får indholdet 3, mens den mindst betydende, kaldet LSB (Least Significant Byte = mindst vigtige byte), får indholdet 232. Dette vil blive grundigt behandlet, når maskinkodeprogrammeringen begynder i kapitel 3.

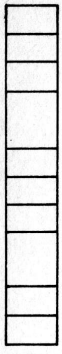
Har du 16 K RAM på din ZX 81, vil byte 16388 indeholde 0, mens byte 16389 vil indeholde tallet 128, og det giver tilsammen adresse 0+256*128 = 32768, der er den første uden for BASIC-systemet.

Den næste del af RAM er den såkaldte programfil, hvor programmer vil blive lagret. Hvordan denne del er opbygget, vil vi vende tilbage til i forbindelse med et program, der omnummererer linierne i programfilen. Programfilen starter på adresse 16509 på ZX81 og 17302 på Lambda.

Den tredje del er udskriftfilen. En begyndelsesadresse kan ikke fastlægges, da det afhænger af, hvor meget programfilen fylder. Udskriftfilen vil nemlig starte umiddelbart efter programfilen. Fylder programmet 1000 bytes, starter udskriftfilen i adresse 17509, fylder det 4000 bytes, begynder filen i adresse 20509. For at finde udskriftfilen i de enkelte tilfælde har vi imidlertid en systemvariabel, der fylder to bytes, og udskriftfilens første adresse findes således med

```
PRINT PEEK 16396+256*PEEK 16397
```

Udskriftfilen er en kopi af skærmbilledet, og den er opbygget efter et helt bestemt mønster.



PEEK 16396+256*PEEK 16397	indeh: 118 NEWLINE
PEEK 16396+256*PEEK 16397+1	1. tegn på skærmen
PEEK 16396+256*PEEK 16397+2	2. tegn på skærmen
–	
–	
PEEK 16396+256*PEEK 16397+32	32. tegn på skærmen
PEEK 16396+256*PEEK 16397+33	indeh: 118 NEWLINE
PEEK 16396+256*PEEK 16397+34	33. tegn på skærmen
–	
–	
PEEK 16396+256*PEEK 16397+791	768. tegn på skærmen
PEEK 16396+256*PEEK 16397+792	indeh: 118 NEWLINE

Som det ses på skitsen starter udskriftfilen med et NEWLINE. Derefter står hele 1. linie fra skærmen, efterfulgt af NEWLINE, hele 2. linie, NEWLINE osv. På hver linie findes 32 tegn, så i alt fylder udskriftfilen 1 + 24 x 33 = 793 bytes. Der er i alt 24 linier på skærmen.

Her skal det bemærkes, at hvis RAMTOP er sat ned under adresse 19711, vil udskriftfilen kun indeholde tegn, der ikke er mellemrum. Derfor vil en CLS-kommando i BASIC med RAMTOP under 19711 gå væsentligt hurtigere, da ZX 81 kun behøver at viske tegn ud, der ikke er mellemrum. Ved CLS med RAMTOP over 19711 vil mellemrum også blive visket ud.

På Lambda er opbygningen lidt anderledes. Udskriftfilen er fastanbragt fra adresse 16509, men opbygningen er den samme. Med udskriftfilens længde på 793 bytes vil næste RAM-område starte på 17302, og her ligger programfilen. Forskellen er altså blot en ombytning af de to filer. Dog vil Lambdas udskriftfil *altid* – RAMTOP's værdi underordnet – fylde 793 bytes. Efter programfilen står en byte med indholdet 255.

Den tredje systemvariabel, der vil blive grundigt omtalt her, er den, der indeholder adressen på den næste printposition; hermed menes det sted, hvor der vil blive udskrevet et tegn næste gang. Med

```
PRINT PEEK 16398+256*PEEK 16399
```


findes positionen, mens

```
PRINT PEEK (PEEK 16398+256*PEEK 16399)
```

naturligvis giver koden på det tegn, der er på det pågældende sted.

Udskriftfilen er det tredje område i RAM. Det fjerde er variabelfilen, og her gemmes de brugte variable og deres værdier. 1. adresse i denne fil findes med

```
PRINT PEEK 16400+256*PEEK 16401
```

Denne kommando

```
PRINT PEEK 16404+256*PEEK 16405
```

vil give adressen på den såkaldte E-linie, der er den første adresse efter variabelfilen.

I området kaldet »linie + syntakskontrol« findes den bearbejdede linie og lidt plads til syntakskontrol af denne. I beregnerstakken foretages alle udregninger. Reserveområdet bliver ikke benyttet. Maskinstakken lagrer nogle returadresser, såvel som GOSUB-stakken lagrer de adresser, som der vendes tilbage til ved brug af RETURN i BASIC.

Kapitel 2

Der findes forskellige talsystemer. Normalt arbejder vi i det såkaldte decimale system, hvor vi tæller enere, tiere, hundreder osv. Imidlertid kan en mikroprocessor ikke arbejde decimalt. Den kan kun arbejde binært. Dermed menes, at den kan arbejde i totalsystemet, hvor der tælles enere, toere, firere, ottere osv. Værdien af et ciffer vil enten være 0 eller 1. Opbygningen ses tydeligt i følgende tabel:

Decimal-systemet (Titals-systemet)	Binært system (Totals-systemet)	Heksadecimal-systemet (Sekstentals-systemet)
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
osv.		

Til højre har vi et tredje system, nemlig det heksadecimale talsystem. Her tælles ligesom i det decimale system op til 9, men så fortsættes med A, B op til F, hvorefter to cifre træder i kraft med 10, 11, 12, 13 osv. Heksadecimale tal er anvendelige i forbindelse med maskinkodeprogrammering, idet et heksadecimalt ciffer svarer til fire binære cifre. Overblikket vil derved blive forbedret.

ZX 81 er en såkaldt 8 bits-computer. Hermed menes, at maskinens lager er delt op i enheder på hver 8 bits. En bit kan antage enten værdien 0 eller værdien 1. Som nævnt i kapitel 1 vil en byte indeholde et tal mellem 0 og 255. 255 svarer til 11111111 i binær notation, nemlig

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

8 bits kan altså udtrykke en værdi mellem 0 og 255, og opbygningen af de otte bits ser således ud:

Værdi	128	64	32	16	8	4	2	1
Bit	7	6	5	4	3	2	1	0

Man har givet de 8 bits forskellige numre. Den mindst betydende bit, den længst til højre, der tæller enere, hedder bit 0, mens den mest betydende bit af de 8 hedder bit 7. Den tæller 128'ere, som det ses på tegningen.

Eks.:
10101010 = 128+32+8+2 = 170

Heksadecimalt svarer dette til de fire første cifre heksadecimalt + de fire sidste heksadecimalt.

Det bliver 1010 = A
og
1010 = A

Heks-koden, som det heksadecimale tal kaldes, er derfor AA for det decimale tal 170.

Som nævnt i kapitel 1 kan man sammensætte to bytes, så de tilsammen kan danne værdier mellem 0 og 65535. Så vil den 1. byte tælle 256'ere, den anden 1'ere.

```
1001   0001   0100   1110   binært
 9      1      4      E     heksadecimalt
   145           78
         37198
```

37198 er netop 256 x 145 + 78, og i heksadecimal notation 914E.

For ikke at forvirre vil vi under maskinkodeprogrammeringen, hvis der opstår tvivl om, hvilket talsystem der benyttes, bruge bogstavet h efter et heksadecimalt tal og b efter et binært tal.

Kapitel 3

I dette kapitel starter vi på maskinkodeprogrammeringen, og vi vil lave mange programmer undervejs. Disse skal tastes ind i din datamat i heksadecimal notation, men vi må først lave et program, der oversætter fra det heksadecimale system til det decimale. Hertil bruges en såkaldt hex-loader:

```
10 LET X = 30000
20 LET A$ = ""
30 IF A$ = "" THEN INPUT A$
40 IF A$ = "S" THEN STOP
50 POKE X,16*CODE A$+CODE A$(2)-476
60 LET X = X+1
70 LET A$ = A$(3 TO)
80 GOTO 30
```

Vi råder dig til at SAVE'e heks-loaderen på bånd, så du kan finde den hurtigt frem. Vi bruger den næsten hele tiden.

I de følgende programmer er det kun heks-koderne til venstre, der skal indtastes. De skal tastes ind parvis, altså 2, 4, 6 eller flere cifre ad gangen, gerne på en lang række. Afslut med S.

Alle maskinkoder har fået et navn, en såkaldt mnemonic, så vi bedre kan huske dem og bruge dem i maskinkodens betydning. Hvis du har købt en såkaldt assembler, kan disse mnemonics tastes direkte ind. De bliver så oversat til maskinkode, dvs. binære tal. Ved indtastningen på heks-loaderen indtastes blot de hekskoder, der står anført i programmet.

Ikke alle adresser er ens på Lambda og ZX 81. Når der ikke er overensstemmelse, vil heks-kode og mnemonic for Lambda stå i parentes under ZX 81-instruktionen, der står normalt. Lambdaejere skal altså erstatte heks-koden og mnemonic'en ovenfor med den i parentes anførte.

I Z80A-assemblersproget har man syv registre, kaldet A, B, C, D, E, H og L. Disse syv registre kaldes 8 bits-registre, og de kan derfor indeholde et tal mellem 0 og 255 (i binær notation mellem 00000000 og 11111111). A-registeret bliver kaldt akkumulatoren, da mange regneinstruktioner opererer på A-registeret. Registrene B og C kan sættes sammen og kaldes et 16-bits-register, BC. Det kan også gøres med DE og HL. Herved opstår 3 16-bits-registre + 1 8 bits-register (A-registeret). Værdien i BC-registeret er lig 256 * værdien i B + værdien i C. Hvis B indeholder 10 og C 40, indeholder BC 256*10+40 = 2600. Det samme gælder med DE og HL. Disse registre kan gives en værdi med instruktionen LD (LoaD). A-registeret kan således få værdien 120 med

```
LD A, 120
```

I skrivemåden benyttes komma i stedet for lighedstegn. Tilsvarende kan B, C, D, E, H og L-registrene LOAD'es. Disse instruktioner fylder hver to bytes. Den første byte bruges til navnet på instruktionen, den næste til værdien, der skal LOAD'es.

```
LD A,120
```

har følgende maskinkode skrevet i det heksadecimale system:

```
3E78 (78h = 120)
```

Heks-koden for instruktionen LD A er 3E. Tallet 120 er som nævnt 78h, så hele instruktionen hedder altså 3E78.

Det var den første mnemonic, du lærte. De er alle opført i appendiks A.

Som nævnt kan de 6 andre 8 bits-registre LOAD'es med en værdi på samme måde. Der findes altså 7 instruktioner, der fungerer på denne måde. Deres koder er:

Heks-kode	**Mnemonic**
3E	LD A,
06	LD B,

0E	LD C,
16	LD D,
1E	LD E,
26	LD H,
2E	LD L,

Der findes også kommandoer til at LOAD'e værdier ind i de tre 16 bits-registre:

Heks-kode	**Mnemonic**
01	LD BC,
11	LD DE,
21	LD HL,

Disse instruktioner fylder 3 bytes: Ovenstående koder og et 16 bits-tal, der fylder to bytes. Når man LOAD'er et 8 bits-register, er mnemonic'en i appendiks B angivet som f.eks.:

```
LD A,n
hvor n er et tal mellem 0 og 255
```

Ved 16 bits-registre, f.eks.:

```
LD BC,nn
hvor nn er et tal mellem 0 og 65535
```

Endnu et eksempel.

```
LD BC,40000
```

40000 kan også skrives som 256*156+64

156 = 9Ch og 64 = 40h. Heks-koden for LD BC,nn er 01h, og hele instruktionens heks-kode er derfor:

```
01 40 9C
```

Bemærk, at den mest betydende byte, 9Ch, angives til sidst – faktisk på samme måde som ved systemvariablene i kapitel 1. Imidlertid er det vigtigt at huske, at det er B-registeret, der LOAD'es med 9Ch, den mest betydende del, og C-registeret, der LOAD'es med den mindst betydende byte, 40h.

Et program i maskinkode kaldes en rutine, og den startes ved hjælp af funktionen USR. Man taler om, at den »kaldes«. Det kan man gøre på flere måder. Når man bruger USR X (hvor X er et tal mellem 0 og 65535, der udtrykker den adresse, hvorfra man starter), får register BC værdien X. Når rutinen engang vender tilbage til BASIC, returneres værdien af BC-registeret.

Når man skriver (i BASIC):

```
PRINT USR X
```

vil ZX 81 udskrive værdien af BC, når rutinen er færdig. Ønskes ingen udskrift, bruges:

```
RAND USR X
```

Disse to metoder vil vi benytte fremover i praksis.

Med dette kan vi nu vise, at vi LOAD'er BC med LD-instruktionen. Prøv følgende program:

Heks-kode	Mnemonic	Forklaring
016400	LD BC, 100	Giv BC-registeret værdien 100
C9	RET	Returner til BASIC

RET-instruktionen betyder bare, at rutinen er slut, og der kan vendes tilbage til BASIC. Hvad RET i virkeligheden gør, får du at vide i et senere kapitel. Som du ser, er heks-koden for instruktionen C9.

På heks-loaderen kan du indtaste:

```
»016400C9«
»S«
```


men du kan også vælge:

```
»016400C9S«
```

blot du husker, at de indtastede koder skal være sat sammen to og to.

Nu ligger dette lille program på adresserne 30000, 30001, 30002 og 30003, fordi heks-loaderen jo starter med at lagre på plads nr. 30000. Husk, at to heksadecimale cifre fylder *én* byte. Skriv dernæst

```
PRINT USR 30000
```

Dette resulterer forhåbentlig i, at ZX 81 skriver 100 (indholdet i BC-registeret) på skærmen.

Man kan LOAD'e en værdi fra ét register til et andet ved brug af følgende instruktioner:

LD	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

Skemaet læses på flg. måde: den lodrette række læses først, den vandrette dernæst. Indholdet i det sidstlæste register bliver kopieret i det førstlæste.

Eksempel:
Mnemonic'en LD A,B har heks-koden 78, og register B's indhold kopieres i A-registeret, men samtidig forbliver værdien også i B.

Læg mærke til, at så unyttige instruktioner som LD A,A og LD B,B etc. findes.

Prøv dette lille program:

Heks-kode	Mnemonic	Forklaring
21D007	LD HL, 2000	Læg 2000 i HL
44	LD B,H	Kopiér HL i BC
4D	LD C,L	
C9	RET	Returner til BASIC

Skriv PRINT USR 30000. Dette giver 2000 som udskrift (indholdet af BC-registeret). Hvorfor bruger vi ikke LD BC,HL? Det er fordi, det kun er 8 bits-registre, som kan udveksle data med hinanden. BC og HL er 16 bits-registre, og de må derfor overføre data som to 8 bits-overførsler.

Vi kan trække en værdi på en givet adresse ind i A-registeret med instruktionen:

LD A,(nn)
hvor nn er et tal mellem 0 og 65535.

Som eksempel kan vi bruge instruktionen:

LD A,(16000)

Læg mærke til parentesen, der bestemmer, at det, vi LOAD'er i A, ikke er 16000, men indholdet på adresse 16000! Naturligvis ville det også være umuligt at LOAD'e A-registeret med 16000, da A er et 8 bits-register! Heks-koden for LD A,(nn) er 3A. 16000 = 128 + 256*62. 128 er 80h og 62 er 3Eh, så den samlede heks-kode for instruktionen er 3A803Eh. Igen er den mest betydende byte til sidst!

Prøv dette eksempel på din heks-loader:

Heks-kode	Mnemonic	Forklaring
3A0714	LD A,(5127)	Læg værdien på 5127 i A
4F	LD C,A	Kopiér A i C
0600	LD B,0	Nulstil B-registeret
C9	RET	Returner til BASIC

Skriv PRINT USR 30000, og resultatet bliver 64 på ZX 81 og 235 på Lambda. Hvis du i BASIC PEEK'er 5127 – der jo ligger i ROM – fås også værdien 64 (235), og vi har dermed bevist gyldigheden af vort udtryk.

Hvorfor LOAD'es B med 0? Fordi PRINT USR, som vi kalder rutinen med, udskriver BC, eller rettere 256*B+C. Da B LOAD'es med 0 og C med A, bliver 256*0+A udskrevet, og vi ønskede jo netop at få udskrevet A: Værdien på adresse 5127.

Der findes også instruktionen LD HL,(nn), hvor nn er en adresse mellem 0 og 65535. Vi kan f.eks. skrive

LD HL,(16396)

Dette vil lægge værdien

PEEK 16396+256*PEEK 16397

ned i HL-registeret. Med denne ordre finder man, som du måske husker fra kapitel 1, udskriftfilen på ZX 81. På Lambda vil det give 1. adresse i programfilen, der altid er 17302. Læg mærke til, at man med LD A,(nn) kun får den værdi, der er lagret på adressen nn, mens LD HL,(nn) giver PEEK nn+256*PEEK (nn+1). Det er selvfølgelig, fordi HL kan rumme 16 bits-tal. Det samme er gældende for BC og DE.

Man kan også få værdien på en adresse udtrykt i HL ned i A-registeret. Det skrives

LD A,(HL)

Hvis HL indeholder 5127, og man derefter udfører LD A,(HL), vil A-registeret indeholde PEEK 5127, som vi fandt værdien af før. Vi prøver det i et eksempel:

Heks-kode	Mnemonic	Forklaring
210714	LD HL,5127	Læg værdien 5127 i HL
7E	LD A,(HL)	Giv A værdien på adresse 5127

0600	LD B,0	Nulstil B
4F	LD C,A	Kopier A i C
C9	RET	Returner til BASIC

PRINT USR 30000 giver forhåbentlig 64 (235). Bemærk, at der ikke er parenteser omkring 5127 i ordren LD HL,5127.

Imidlertid kan værdien på adresse HL lægges ned i alle de nævnte 8 bits-registre, så vi kan effektivisere vort program som følger:

Heks-kode	Mnemonic	Forklaring
	LD HL,5127	Læg værdien 5127 i HL
	LD B,0	Nulstil B
	LD C,(HL)	Giv C værdien på adresse 5127
	RET	Returner til BASIC

Find selv heks-koderne.

I alt findes der disse instruktioner:

Heks-kode	Mnemonic
7E	LD A,(HL)
46	LD B,(HL)
4E	LD C,(HL)
56	LD D,(HL)
5E	LD E,(HL)
66	LD H,(HL)
6E	LD L,(HL)
0A	LD A,(BC)
1A	LD A,(DE)
ED4B	LD BC,(nn)
ED5B	LD DE,(nn)
2A	LD HL,(nn)

Læg mærke til, at LD BC,(nn) og LD DE,(nn) kommer til at fylde 4 bytes, da tallet nn jo fylder 2 bytes.

Som du måske har opdaget, hvis du har kigget i appendiks A,

så står mnemonics opført i flere grupper. Den første gruppe består af de instruktioner, der bruges uden omsvøb. Under dem står en gruppe, der skal have CBh foran sig. En tredje gruppe skal have EDh foran. LD BC,(nn) har heks-koden ED4B. Den er altså én af de instruktioner, der behøver en kode foran. 4Bh giver instruktionen LD C,E – og det er jo noget helt andet.

Vi kan vise brugen i et eksempel:

Heks-kode	Mnemonic	Forklaring
ED4BE803	LD BC,(1000)	Giv BC værdien PEEK 1000 + 256*PEEK 1001
C9	RET	Returner til BASIC

PRINT USR 30000 giver svaret 13867 på ZX 81 og 10296 på Lambda. Du kan jo kontrollere svarets rigtighed i BASIC med netop PEEK 1000+256*PEEK 1001

Vi kan også i maskinkode lægge en værdi ind på en given adresse. Skrives:

```
LD (29998),A
```

vil værdien i A-registeret blive lagt på adresse 29998. Læg igen mærke til parentesen, der fortæller, at vi behandler en adresse og ikke en værdi. Denne og de andre instruktioner, der lægger værdier på adresser, skal behandles med samme omhu som POKE i BASIC. Hvis der går noget galt, er *crash* den mest sandsynlige udgang.

Koden for LD (nn),A er 32h plus de 2 bytes, som angiver adressen. Vi kan afprøve dette med et program:

Heks-kode	Mnemonic	Forklaring
3E64	LD A,100	LOAD A-reg med 100
322E75	LD (29998),A	Læg A-reg's indhold på adresse 29998
C9	RET	Returner til BASIC

Vi udfører rutinen med RAND USR 30000. Som nævnt tidligere, vil rutinen blot blive udført, uden at BC-registerets indhold skrives på skærmen. Det er vi ligeglade med her. Når du har kørt rutinen, skriver du PRINT PEEK 29998, hvilket gerne skulle give A-registerets indhold (tallet 100), der jo er overført til denne adresse.

Som du måske har gættet, så findes der også instruktioner, der lægger 16 bits-tal på to adresser. De hedder

Heks-kode	Mnemonic
ED43	LD (nn),BC
ED53	LD (nn),DE
22	LD (nn),HL

Vi prøver et eksempel:

Heks-kode	Mnemonic	Forklaring
011027	LD BC,10000	Læg 10000 i BC-registeret
ED432E75	LD (29998),BC	Værdien af C lægges på 29998 Værdien af B lægges på 29999
C9	RET	Returner til BASIC

Som det ses i forklaringen vil LD (29998),BC lægge C-registerets indhold på den første plads, 29998, og B-registerets indhold på den anden plads, 29999. Husk, at B er det mest betydende register (det, der tæller 256'ere) i BC-registeret, og det lagres altså igen sidst! Start rutinen med

RAND USR 30000

og kontrollér med PRINT PEEK 29998+256*PEEK 29999. Svaret skulle gerne være 10000. Hvis du prøver med PEEK 29998 og PEEK 29999 hver for sig, får du først indholdet af C-registeret, dernæst B-registeret.

Der findes også instruktioner, der lægger et tal fra et register

på en adresse specificeret i et andet register. Disse, og de før omtalte ordrer, er opført her:

Heks-kode	Mnemonic
77	LD (HL),A
70	LD (HL),B
71	LD (HL),C
72	LD (HL),D
73	LD (HL),E
74	LD (HL),H
75	LD (HL),L
36	LD (HL), n, hvor n er et tal mellem 0 og 255
02	LD (BC),A
12	LD (DE),A
32	LD (nn),A, hvor n er et tal mellem 0 og 255

Vi prøver med et eksempel:

Heks-kode	Mnemonic	Forklaring
212E75	LD HL,29998	Giv HL-reg indholdet 29998
3E64	LD A,100	Giv A-reg indholdet 100
77	LD (HL),A	Læg A-reg's værdi på adresse HL
C9	RET	Returner til BASIC

RAND USR 30000. Prøv med PRINT PEEK 29998. Svaret skal være 100. Husk, at der her kun bliver lagt ind på adresse HL og *IKKE* også på adresse (HL+1).
Prøv også dette:

Heks-kode	Mnemonic	Forklaring
212E75	LD HL,29998	Giv HL-reg indholdet 29998
3A3800	LD A,(56)	Læg indholdet på adresse 100 i A
77	LD (HL),A	Læg A-reg's værdi på adresse HL

C9 RET Returner til BASIC

RAND USR 30000. Svaret ved PRINT PEEK 29998 skal være 13, da der på adresse 56 (i ROM) står koden 13.

Her er en næsten komplet oversigt over LD-kommandoer. Skemaet vil blive suppleret senere.

8-bit LOAD

LD	A	B	C	D	E	H	L	(HL)	BC)	(DE)	(nn)
A	7F	78	79	7A	7B	7C	7D	7E	0A	1A	3A
B	47	40	41	42	43	44	45	46	–	–	–
C	4F	48	49	4A	4B	4C	4D	4E	–	–	–
D	57	50	51	52	53	54	55	56	–	–	–
E	5F	58	59	5A	5B	5C	5D	5E	–	–	–
H	67	60	61	62	63	64	65	66	–	–	–
L	6F	68	69	6A	6B	6C	6D	6E	–	–	–
(HL)	77	70	71	72	73	74	75	–	–	–	–
(BC)	02	–	–	–	–	–	–	–	–	–	–
(DE)	12	–	–	–	–	–	–	–	–	–	–
nn	32	–	–	–	–	–	–	–	–	–	–

16-bit LOAD

LD	BC	DE	HL	nn	(nn)
BC	–	–	–	01	ED4B
DE	–	–	–	11	ED5B
HL	–	–	–	21	2A
(nn)	ED 43	ED 53	22	–	–

Inden vi går i gang med flere instruktioner, vil vi præsentere endnu et register i Z80A, nemlig det såkaldte flagregister, F-registeret!

Flagregisteret har kun mening i binær notation, da hver bit

har sin egen opgave. Hver bit har endvidere fået navn, så man bedre kan kende forskel på dem.

Registeret er bygget således op:

Flagregisteret	S	Z		H		P/V	N	C
Bitnummer	7	6	5	4	3	2	1	0

Som det ses på tegningen har bit 3 og bit 5 ingen navne. De har kun intern betydning. Flagene H og N, bit 4 og bit 1, har kun betydning for DAA-instruktionen, som vi omtaler senere. Tilbage har vi fire flag. En grundig præsentation vil vi ikke give her, men under den videre gennemgang af instruktionerne vil vi se på, hvordan de påvirker flagene.

Det første flag, bit 0 i F-registeret, er C-flaget, det såkaldte *carry-flag*. Dette flag sættes, dvs. får værdien 1, hvis en sum for et 8 bits-register fylder mere end 8 bits, hermed menes større end 255, eller hvis en sum for et 16 bits-register er større end 65535 og dermed fylder mere end 16 bits. Carry-flaget fungerer altså som en slags mente.

Det næste flag er Z-flaget, det såkaldte *zero-flag*. Dette flag sættes, når en operation på et register medfører nul som resultat.

S-flaget er det såkaldte *sign-flag*. Dette flag kopierer »sign bit« i et resultat. Sign bit er den ottende bit (bit 7) i et 8 bits-register eller sekstende bit i et 16 bits-register.

Det fjerde og sidste flag er P/V-flaget. P/V står for *Parity/overflow*. Som navnet antyder, har dette flag to opgaver. Efter aritmetiske instruktioner virker flaget som *overflowindikator*, hvor det sættes, hvis et resultat har forkert fortegn. Efter bestemte instruktioner virker flaget som *paritetsindikator*, dvs. hvis et resultat medfører et lige antal ettaller, hermed menes 2, 4, 6, 8, 10, 12, 14 eller 16 ettaller i binær notation, sættes flaget.

De fire sidstomtalte flag er meget vigtige, og de er ydermere testbare. Deres tilstand (0 eller 1) kan afgøre, hvordan et program skal forløbe, men dette vil vi også vende tilbage til.

Der er ingen instruktion, såsom LD F,78, til registeret. Z80 bestemmer selv, hvilken værdi flagregisteret indeholder.

Det er værd at bemærke, at ingen af de beskrevne LOAD-operationer påvirker flagene.

Kapitel 4

Man har også i maskinkode brug for at lægge til og trække fra, og vi har derfor bestemte instruktioner dertil. INC (INCrement) lægger én til værdien af et register, og DEC (DECrement) trækker tilsvarende én fra værdien. Instruktionerne opererer både på 8 og 16 bits-registre. De fylder kun en byte, og der findes følgende:

Heks-kode	Mnemonic	Heks-kode	Mnemonic
3C	INC A	3D	DEC A
04	INC B	05	DEC B
0C	INC C	0D	DEC C
14	INC D	15	DEC D
1C	INC E	1D	DEC E
24	INC H	25	DEC H
2C	INC L	2D	DEC L
34	INC (HL)	35	DEC (HL)
03	INC BC	0B	DEC BC
13	INC DE	1B	DEC DE
23	INC HL	2B	DEC HL

Der er således 16 8 bits- og 6 16 bits-ordrer. Med den nuværende viden kan vi lave et program, som udskriver på skærmen:

Heks-kode	Mnemonic	Forklaring
2A0C40	LD HL,(16396)	Udskriftfilen lokaliseres i HL
(217D40 – LD HL,16509)		(Kode til Lambda)
23	INC HL	1. punkt på skærmen
363F	LD (HL),63	Udskriv CHR$ 63
23	INC HL	Næste punkt på skærmen
3624	LD (HL),36	Udskriv CHR$ 36
23	INC HL	Næste punkt på skærmen

361C	LD (HL),28	Udskriv CHR$ 28
23	INC HL	Næste punkt på skærmen
3626	LD (HL),38	Udskriv CHR$ 38
C9	RET	Returner til BASIC

Start programmet med RAND USR 30000.

Vi vil gerne her gøre opmærksom på, at vores Lambda-program kan forbedres. På denne datamat ligger udskriftfilen fast, som før beskrevet, fra adresse 16509. De to første instruktioner kunne slås sammen til LD HL,16510. For at vores programforklaringer kan være brugbare til begge datamater, vil programmer til Lambda være opbygget på samme måde som til ZX 81.

Instruktionen INC (HL) kan bruges således:

Heks-kode	Mnemonic	Forklaring
212E75	LD HL,29998	Læg 29998 ind i HL
3664	LD (HL),100	Læg værdien 100 på adresse 29998
34	INC (HL)	Værdien på adresse 29998 incrementeres
C9	RET	Returner til BASIC

RAND USR 30000.

INC (HL) kræver en nærmere forklaring. Denne instruktion finder ud af, hvilken adresse HL peger på, i vores tilfælde adresse 29998. Indholdet på denne adresse forøges med én. I dette tilfælde ændres adressens indhold fra 100 til 101. Prøv selv, hvis du ikke allerede har gjort det, med PEEK 29998. HL indeholder stadig 29998.

8 bits-INC/DEC påvirker Z-flaget, der sættes, hvis resultatet bliver nul. Hvis A f.eks. har indholdet 2, og der decrementeres to gange, vil Z-flaget blive sat, da A da er talt ned til nul. Du undrer dig sikkert over, at resultat kan blive nul, når man adderer én. Det kan det dog. Hvis A indeholder 255, og INC A udføres, starter A simpelt hen forfra, da det ikke kan tælle til mere end 255. Z-flaget sættes.

P/V-flaget virker ved 8 bits-INC/DEC som overflowindikator, hvilket beskrives under 2-komplement-metoden i næste kapitel.

S-flaget sættes/resættes afhængigt af resultat, men det kopierer altså stadig blot bit 7.

C-flaget påvirkes ikke af INC eller DEC.

Ved 16 bits-INC/DEC påvirkes *ingen* flag.

Vi har også brug for instruktioner til at lægge mere til eller trække mere fra end én. Til dette formål bruges de fire instruktioner ADD (ADDition), ADC (ADdition with Carry), SUB (SUBtraction) og SBC (SuBtraction with Carry). Der findes følgende instruktioner:

Heks-kode	Mnemonic	Heks-kode	Mnemonic
80	ADD A,B	88	ADC A,B
81	ADD A,C	89	ADC A,C
82	ADD A,D	8A	ADC A,D
83	ADD A,E	8B	ADC A,E
84	ADD A,H	8C	ADC A,H
85	ADD A,L	8D	ADC A,L
86	ADD A,(HL)	8E	ADC A,(HL)
87	ADD A,A	8F	ADC A,A
C6	ADD A,n	CE	ADC A,n
09	ADD HL,BC	ED4A	ADC HL,BC
19	ADD HL,DE	ED5A	ADC HL,DE
29	ADD HL,HL	ED6A	ADC HL,HL
90	SUB A,B	98	SBC A,B
91	SUB A,C	99	SBC A,C
92	SUB A,D	9A	SBC A,D
93	SUB A,E	9B	SBC A,E
94	SUB A,H	9C	SBC A,H
95	SUB A,L	9D	SBC A,L
96	SUB A,(HL)	9E	SBC A,(HL)
97	SUB A,A	9F	SBC A,A
D6	SUB A,n	DE	SBC A,n
		ED42	SBC HL,BC
		ED52	SBC HL,DE
		ED62	SBC HL,HL

Instruktionen skal forstås således, at indholdet i registeret efter kommaet adderes til hhv. subtraheres fra indholdet i registeret før kommaet.

Eksempel:

ADD A,B	Adder værdien af B til værdien i A
ADD A,(HL)	Adder indholdet på adresse HL til værdien i A
ADD A,n	Adder et tal n mellem 0 og 255 til værdien i A
ADD HL,BC	Adder værdien i BC til værdien i HL

På tilsvarende måde virker SUB, men der trækkes naturligvis fra. Blandt instruktionerne er der en ordre, der fordobler A-registerets indhold: ADD A,A.

ADC-instruktionen kræver en nøjere forklaring. Instruktionen skal forstås på den måde, at hvis C-flaget, carry-flaget, er sat, tillægges yderligere én.

Eksempel:

ADC A,10: adder 10 + værdien af C-flaget til værdien i A

SBC-instruktionen virker på samme måde, blot vil værdien af carry-flaget blive fratrukket.

Eksempel:

SBC A,10: træk 10 + værdien af C-flaget fra værdien i A

Som du måske har bemærket, så findes der kun 16 bits-addition, der opererer på HL-registeret. Hvis vi ønsker at addere DE til BC og gemme summen i HL, så må vi benytte en anden metode. Hvis BC indeholder 1000 og DE 651, så skulle resultatet gerne blive 1651. Det ser således ud binært:

BC (1000):	00000011	11101000
DE (651):	00000010	10001011
HL (1651):	00000110	01110011

Vi kan anvise to forskellige muligheder:

Eksempel a:

Heks-kode	Mnemonic	Forklaring
79	LD A,C	Kopiér C-reg's indhold i A
83	ADD A,E	Læg E-reg's indhold til
6F	LD L,A	Summen lægges i L (LSB færdiglagt)
78	LD A,B	Kopiér B-reg's indhold i A
8A	ADC A,D	Læg D-reg's indhold + en evt. mente ved den første addition til
67	LD H,A	Summen lægges i H (MSB færdiglagt)

Men det kan gøres hurtigere – eksempel b:

Heks-kode	Mnemonic	Forklaring
60	LD H,B	Kopiér B-reg's indhold i H
69	LD L,C	Kopiér C i L. BC er nu kopieret i HL
19	ADD HL,DE	Tillæg DE til BC – summen ligger nu i HL

Vi benytter her metode b i et eksempel:

Heks-kode	Mnemonic	Forklaring
01E803	LD BC,1000	LOAD BC med 1000
118B02	LD DE,651	LOAD DE med 651
60	LD H,B	Kopiér B-reg's indhold i H
69	LD L,C	Kopiér C-reg's indhold i L
19	ADD HL,DE	Adder DE til BC – summen i HL
44	LD B,H	Kopiér H-reg's indhold i B
4D	LD C,L	Kopiér L-reg's indhold i C
C9	RET	Returner til BASIC

PRINT USR 30000. Resultatet af additionen skulle gerne blive 1651. I vor forklaring til ADD HL,DE har vi skrevet, at DE tillægges BC, men naturligvis lægges DE til HL. Vi har blot hentet BC ned i HL kort forinden, så i praksis fungerer instruktionen således. BC indeholder stadig 1000 efter additionen, og DE 651.

Hvis vi ønsker at trække DE fra BC og lægge differensen i HL, skal vi foretage den modsatte operation. Vi kan atter anvise to metoder. Den første ser således ud:

Heks-kode	Mnemonic	Forklaring
79	LD A,C	Hent C-reg's indhold op i A
93	SUB A,E	Træk E-reg's indhold fra A (C)
6F	LD L,A	Differensen lægges i L (LSB færdig)
78	LD A,B	B-reg's indhold lægges op i A
9A	SBC A,D	D-reg's indhold + evt. carry trækkes fra
67	LD H,A	Differensen lægges i H (MSB færdig)

Vi vil bruge dette i et eksempel. Vi LOAD'er BC med 882, og DE med 169. Resultatet ved en subtraktion skulle gerne blive 713. Vi kigger på det i binær notation:

BC (882):	00000011	01110010
DE (169):	00000000	10101001
HL (713):	00000010	11001001

Forklar dig selv fremgangsmåden ved den binære subtraktion. Du starter yderst til højre: 0-1 er 1 og 1 i mente, og derfor vil den næste bit blive 0 (1-1). Bit 8 i HL-registeret (bit 0 i H) bliver 0, da der er blevet sat et carry-flag ved subtraktionen på bit 7. Med andre ord: carry-flaget sættes, når LSB af det register, der skal trækkes fra, er større end det register, hvorfra der skal trækkes.

Vi bruger denne metode i et eksempel:

Heks-kode	Mnemonic	Forklaring
017203	LD BC,882	LOAD BC-registeret med 882
11A900	LD DE,169	LOAD DE-registeret med 169
79	LD A,C	
93	SUB A,E	
6F	LD L,A	Subtraktionen foretages
78	LD A,B	

9A	SBC A,D	
67	LD H,A	
44	LD B,H	Kopiér H i B
4D	LD C,L	Kopiér L i C
C9	RET	Returner til BASIC

PRINT USR 30000. Resultatet skulle gerne være 713. Vi har i vores eksempel lagt resultatet af subtraktionen i HL. Derefter er det lagt over til udskrivning i BC. Resultatet kunne også være lagt i BC med det samme, og overførslen havde været unødvendig.

Nu har vi gennemgået additions- og subtraktionsinstruktionerne. Bemærk i øvrigt, at der ikke findes 16 bits-SUB, men kun SUB med carry (SBC). Skulle vi have benyttet metode b i ovenstående eksempel, skulle vi først sikre os et resat carry-flag.

Også ved disse fire instruktioner gælder det, at registrene starter forfra, når de ikke længere kan udtrykke tallet. I eksemplet

```
LD HL,40000
LD BC,34567
ADD HL,BC
```

vil HL rumme resultatet 9031.

På samme måde med

```
LD A,30
SUB 49
```

hvor resultatet bliver 237.

ADD, ADC, SUB og SBC – på nær ADD HL,BC, ADD HL,DE og ADD HL,HL – påvirker flagene på følgende måde:

Carry-flaget sættes, hvis en 9. hhv. 17. bit bliver nødvendig. 40000+34567 = 9031, som vi så før, vil sætte flaget. 237+49 = 30 vil også sætte det.

Z-flaget sættes – som sædvanlig – hvis resultatet bliver nul.

S-flaget kopierer sign bit, og P/V-flaget virker som overflow-indikator, som vi ser på i det følgende kapitel.

Undtagelserne – ADD med HL-registeret – påvirker kun C-flaget.

Vi vil afslutte dette kapitel med et eksempel på brugen af disse instruktioner. Dette program vil udskrive øverst på skærmen, hvor mange bytes et BASIC-program inkl. variable fylder.

Rutinen ser således ud:

Heks-kode	Mnemonic	Forklaring
019643	LD BC,17302	BC får værdien 17302 ZX 81: Startadressen på programfilen tillagt længden af udskriftfilen (16509+793) Lambda: Startadressen på programfilen
2A1440	LD HL,(16404)	HL LOAD'es med 1. adresse efter variabelfilen
ED42	SBC HL,BC	BC trækkes fra HL – og herved findes længden af programfilen + variabelfilen
44	LD B,H	H-reg's indhold kopieres i B
4D	LD C,L	L-reg's indhold kopieres i C
C9	RET	Returner til BASIC

PRINT USR 30000 vil udskrive antallet af brugte bytes. En forklaring på programmet er som vanligt angivet ud for de enkelte mnemonics.

Kapitel 5

I maskinkode er det ikke muligt at LOAD'e registre med negative tal, som man kunne forestille sig, at det kunne ske med

LD A,–10

Denne instruktion findes ikke, men det kan dog lade sig gøre at udtrykke negative tal. Til dette formål bruges den såkaldte *2-komplement-metode.* 2-komplementet findes ved at vende hver enkelt bit, og derefter addere én. Derved får vi følgende tabel:

+127	i binær notation	01111111
+126	–	01111110
+125	–	01111101
...		
+2	–	00000010
+1	–	00000001
0	–	00000000
–1	–	11111111 = 255
–2	–	11111110 = 254
...		
–126	–	10000010 = 130
–127	–	10000001 = 129
–128	–	10000000 = 128

En fuldstændig liste findes i appendiks C.

Som det ses, kan man også finde 2-komplementet ved at trække tallet (det negative tal, der skal udtrykkes) fra 256. –126 udtrykkes derfor som 130. Som du vil opdage, er bit 7 sat, når tallet kan betragtes som negativt. Bit 7 kaldes derfor også »sign bit«, og S-flaget kopierer som allerede nævnt denne bit for at vise, om et tal er negativt.

Eksempel:
Hvis vi ønsker at lægge 130 til –48, kan vi gøre det på følgende måde: –48 er i 2-komplement (256–48) 208. Programmet bliver derfor:

Heks-kode	Mnemonic	Forklaring
3ED0	LD A,208	Betragtes som –48
C682	ADD A,130	Læg 130 til de –48
4F	LD C,A	Overfør til
0600	LD B,0	BC-registeret
C9	RET	Returner til BASIC

PRINT USR 30000. Resultatet bliver 82.

Dette stemmer også overens med det, som vi fortalte ved ADD. Hvis tallet blev for stort, startes forfra igen. Hvad nu, hvis vi lægger 23 til –100.

Heks-kode	Mnemonic	Forklaring
3E9C	LD A,156	Load A med –100 (156)
C617	ADD A,23	Adder 23
4F	LD C,A	Overfør til
0600	LD B,0	BC-registeret
C9	RET	Returner

Resultatet vil være 179, og det er jo forkert. Men bemærk, at bit 7 er sat, så 2-komplementært er svaret rigtigt, og det er netop –77, som det ses i appendiks C.

Specielt ved denne metode er det relevant med en omtale af P/V-flaget som overflowindikator. Hvis addition af to positive tal medfører, at bit 7 bliver sat (summen er over 127), vil det 2-komplementært være negativt, hvilket er forkert. Overflow-flaget sættes som tegn på, at der er sket en fejl. 34 + 104 er 138, der er positivt, men 2-komplementært negativt. P/V-flaget indikerer fejlen.

To negative tal kan medføre samme fejl. Hvis de lægges sammen, og bit 7 resættes, er der sket en fejl. –34 + (–104) er –138,

der er negativt, men 2-komplementært positivt. Atter viser P/V-flaget, at der er noget galt.

To tal med modsat fortegn kan ved addition ikke medføre overflow. Se de fire eksempler her:

1)	34	00100010	
	+104	01101000	
	138	10001010	P/V-flaget sat
2)	−34	11011110	
	+−104	10011000	
	−138	01110110	P/V-flaget sat
3)	34	00100010	
	+−104	10011000	
	−70	10111010	P/V-flaget resat
4)	104	01101000	
	+−34	11011110	
	70	01000110	P/V-flaget resat

Ved subtraktion er det lige modsat. Da vil overflowflaget blive sat ved to tal med modsat fortegn, da differensen udtrykkes forkert 2-komplementært. Et enkelt eksempel:

34	00100010	
−−104	10011000	
138	10001010	P/V-flaget sat

Du må selv bestemme, om du vil bruge 2-komplementsystemet i dine egne beregninger, men helt snige dig udenom kan du ikke, hvilket du vil indse senere.

Hvis man udtrykker tal i 2-komplementmetoden kan man – som det ses i skemaet – kun arbejde med tal imellem −128 og 127 med 8 bits-registre. For 16 bits-registre kan man naturligvis arbejde mellem −32768 og 32767. Du kan selv arbejde videre med 16 bits negative tal.

Som eksempel: −27123 er 38413. I programmer skal du bruge ADC i stedet for ADD, da sidstnævnte instruktion kun sætter C-flaget og ikke S-flaget. Senere kan du få at vide, hvordan du kan resætte C-flaget, så du undgår en eventuel mente.

Z80A-chip'en har endnu to registre, som vi ikke har omtalt. De hedder IX og IY. Disse to er lidt specielle, for det er – for det første – deciderede 16 bits-registre. De kan altså ikke skilles i to 8 bits-registre. I'et står for »indeks«. De er også specielle på en anden måde, fordi man kan bruge f.eks. følgende instruktion:

```
LD B,(IY+22)
```

Hvis IY er 2000, vil B blive LOAD'et med indholdet på adresse 2000 + 22 = 2022. På samme måde kan man trække en konstant fra.

```
LD C,(IX−100)
```

Denne instruktion vil LOAD'e C med indholdet på adresse 16700, hvis IX indeholder 16800. −100 udtrykkes 2-komplementært.

Du skal dog passe lidt på. Hvis du ændrer IX's værdi i SLOW-tilstand, vil datamaten *crashe*. Det samme gælder til dels også for IY. I visse tilfælde kan man dog ændre dette registers værdi i SLOW. IY har en fordel, idet ROM giver det værdien 4000h (16384), når du starter en maskinkoderutine. Fordelen er, at systemvariablene starter netop på dette sted, som du vel ved. Så har man direkte adgang til dem. Du kan ikke bruge LD B,(16420). Den instruktion findes simpelt hen ikke, men ved brug af IY opnås det ønskede med

```
LD B,(IY+36)
```

da 16384 + 36 er 16420.

For at finde heks-koden for instruktioner, hvor IX og IY er involveret, skal du bare lade, som om du bruger HL i stedet, og så finde koden for den instruktion. Foran denne kode skriver du så

DDh eller FDh, afhængig af, om det var IX eller IY. IX-instruktioner begynder med DDh, IY– med FDh. Hvis IX eller IY er i parentes, altså POKE- eller PEEK-instruktioner, er der en konstant tillagt. Denne konstant, som er et tal i 2-komplement-metoden, skal selvfølgelig angives i heks-koden, og den er *altid* 3. byte i denne, selv om det betyder, at koden bliver »delt« i to dele. Eksempel:

Fordi koden for LD (HL),20h er 3620h, så er koden for LD (IY+0),20h altså FD360020h. Læg mærke til, at selv om konstanten er nul, skal den medtages. Du kan se alle heks-koderne og mulighederne med IX og IY i appendiks A. Hvis konstanten er negativ, skal den udtrykkes i 2-komplement-metoden. Eksempel: LD (IY–13),20h bliver FD36F320h.
Prøv dette eksempel:

Heks-kode	Mnemonic	Forklaring
FD2A0C40	LD IY,(16396)	Læg udskriftfilens startadresse i IY
(FD217D40 – LD IY,16509)		
FD360108	LD (IY+1),8	Skriv CHR$ 8 på 1. punkt på skærmen
FD362280	LD (IY+34),128	Skriv CHR$ 128 på 2. linje, 1. punkt
C9	RET	Returner til BASIC

Når rutinen returnerer, sørger ROM for, at IY igen indeholder 4000h. Den bruger nemlig selv systemvariablene meget flittigt.

4 bytes er i øvrigt den længste heks-kode, der findes for Z80A-instruktioner. Flagsætning for en IX- og IY-instruktion er den samme som for den tilsvarende HL-instruktion.

*

Der eksisterer flere 8 bits-registre, og det er to meget specielle registre, kaldet I-registeret (Interrupt-registeret) og R-registeret (Refresh-registeret).

ZX 81 (ikke Lambda) bruger I-registeret til sin displayrutine. Registeret indeholder altid 1Eh, idet tegngeneratoren starter i

ROM på adresse 1E00h. Der er to instruktioner, som benytter I-registeret, nemlig LD A,I og LD I,A. Når du bliver lidt skrappere, kan du bruge I-registeret til oplagring af mellemresultater eller til manipulation af skærmen. Det sidste kan give brugerdefineret højopløsningsgrafik, men det kræver mere viden, end vi vil komme ind på. Vi vil dog gerne demonstrere brugen af I-registeret for dig med følgende program, der kun virker med effekt på ZX 81.

Heks-kode	Mnemonic	Forklaring
3E10	LD A,16	Læg værdien 16 i A-registeret.
ED47	LD I,A	Kopiér A i I-registeret.
C9	RET	Returner til BASIC
3E1E	LD A,30	Læg værdien 30 i A-registeret.
ED47	LD I,A	Kopiér A i I-registeret.
C9	RET	Returner til BASIC

Lav følgende BASIC-program:

```
100 RAND USR 30000
110 PAUSE 200
120 RAND USR 30005
```

Skriv RUN.

Først kaldes den første rutine, hvor 16 lægges i I-registeret. Der returneres, og 4 sekunder senere kaldes den anden rutine, der starter på adresse 30005, og 30, eller 1Eh, lægges i I-registeret. Der returneres.

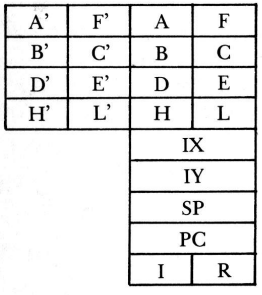
Det skaber et mærkværdigt billede, men efter 4 sekunder kaldes det normale skærmbillede tilbage, da I-registeret skal indeholde værdien 30 for et normalt billede. På Lambda giver ingen værdier problemer. Dette bevirker, at ovenstående program ikke virker efter hensigten på Lambda. Senere bringer vi et lignende program, som vil lave mystiske mønstre på skærmen.

Måske tror du, at I-registeret er en 8 bits-del af IX- og IY-registrene, men denne tanke må du aflive, hvis du har den. I-registeret må ikke forveksles med de to index-registre, da det intet har med dem at gøre.

Om det andet register, R-registeret, kan der ikke siges meget. Dette register tæller én op for hver udført instruktion. Efter 127 nulstilles registeret, og herfra kan man altså generere et forholdsvis tilfældigt tal. Til dette register findes også kun 2 instruktioner, og det er LD A,R og LD R,A. Dette vil du få meget svært ved at bruge i egne rutiner.

LD I,A og LD R,A påvirker som de eneste LOAD-instruktioner flagene. S-flaget kopierer sign bit, Z-flaget sættes, hvis et nul kopieres. P/V-flaget behandler såkaldte interrupts, men det arbejder vi ikke med her. C-flaget berøres ikke.

Her er en samlet oversigt over registrene i Z80:



F = flag-registeret
" ' "viser, at det er et alternativt register (omtales i kapitel 16)
SP = stackpointer (stakpil) (omtales i kapitel 10)
PC = programcounter (programtæller) (omtales i kapitel 7)

Kapitel 6

Og nu til noget helt andet...

Der findes tre logiske instruktioner, der arbejder på de enkelte bits i A-registeret. Disse hedder AND, OR og XOR. To bits sammenlignes, og resultatet afhænger naturligvis af hvilken af de tre ordrer, der er benyttet.

Bruges AND-instruktionen dannes resultatet efter nedenstående sandhedstabel.

0 og 0 sammenlignes – resultatet er 0
1 og 0 sammenlignes – resultatet er 0
0 og 1 sammenlignes – resultatet er 0
1 og 1 sammenlignes – resultatet er 1

Vi kan illustrere det med et eksempel:

register A: 00110101 = 53
AND 163 10100011 = 163
register A: 00100001 = 33

A-registeret indeholder 53 og sammenlignes med 163. Resultatet bliver 33. Bit 0 og bit 5 forbliver 1, da de to sammenlignede bits begge er 1. De andre bits bliver 0, fordi der ikke sammenlignes to ettaller, og ifølge vor sandhedstabel skal BÅDE den ene OG den anden være 1.

Som tommelfingerregel kan det siges, at resultatet aldrig vil blive større end den mindste værdi af de to sammenlignede størrelser i decimal notation.

Vi vil vise AND-instruktionen i et eksempel:

Heks-kode	Mnemonic	Forklaring
3E64	LD A,100	A-registeret LOAD'es med 100
E6C8	AND 200	A-reg's bits sammenlignes med 200
0600	LD B,0	B-registeret får værdien nul
4F	LD C,A	Kopiér A i C-registeret
C9	RET	Returner til BASIC

Anskuet binært ser sammenligningen således ud:

register A: 01100100 = 100
AND 200 11001000 = 200
register A: 01000000 = 64

Skriv PRINT USR 30000. Svaret bør blive 64.

Der findes følgende AND-instruktioner:

Heks-kode	Mnemonic	
A7	AND A	
A0	AND B	
A1	AND C	
A2	AND D	
A3	AND E	
A4	AND H	
A5	AND L	
A6	AND (HL)	
DDA6	AND (IX+d)	
FDA6	AND (IY+d)	
E6	AND n	hvor d er et tal mellem −128 og 127

I oversigten her har vi som det ses også medtaget AND-instruktionen, som opererer på IX- og IY-registrene. Koden bør ikke volde vanskeligheder forståelsesmæssigt.

Vi har kun benyttet sidstnævnte instruktion, men A-registeret kan også sammenlignes med et andet register. Bemærk, at der altid sammenlignes med A-registeret som den ene part; resultatet lægges *altid* i A. Det andet register beholder sin værdi.

Eksempel:

Heks-kode	Mnemonic	Forklaring
3E38	LD A,56	LOAD A-registeret med 56
213800	LD HL,56	LOAD HL-registeret med 56
A6	AND (HL)	Sammenlign A med værdien på adresse HL (56)
0600	LD B,0	LOAD B-registeret med nul
4F	LD C,A	Kopier resultatet af sammenligningen (A) i C
C9	RET	Returner til BASIC

Vi kigger på det binært:

register A: 00111000 = 56
AND (HL) 00001101 = 13 (PEEK 56)
register A: 00001000 = 8

Skriv PRINT USR 30000. Resultatet er forhåbentlig 8.

AND bruges ofte til at resætte enkelte bits i A-registeret. Ønsker vi at nulstille bit 0, 1, 2, 3, 4 kan vi bruge kommandoen AND 11100000b, dvs. AND 224.

AND-instruktionen påvirker flagene på følgende måde: Carry-flaget resættes *altid* efter AND-instruktion, da brug af en 9. bit er umulig via vor tommelfingerregel. Z-flaget sættes, hvis instruktionen medfører nul i A-registeret. S-flaget kopierer traditionen tro sign bit, altså bit 7 i A-registeret efter sammenligningen. P/V-flaget virker som paritetsindikator, hvilket er beskrevet under flagsætningen.

OR-instruktionen er nært beslægtet med AND, blot er resultatet anderledes, da sandhedstabellen for OR ser således ud:

0 og 0 sammenlignes – resultatet er 0
0 og 1 sammenlignes – resultatet er 1
1 og 0 sammenlignes – resultatet er 1
1 og 1 sammenlignes – resultatet er 1

Heraf fremgår det, at hvis blot den ene af de to sammenlignede bits har værdien 1, bliver resultatet 1. Eksempel:

register A: 01100111 = 103
register B: 10100100 = 164
OR B: 11100111 = 231

Vi vil atter vise det i et eksempel via heks-loaderen:

Heks-kode	Mnemonic	Forklaring
3E64	LD A,100	LOAD A-registeret med 100
06C8	LD B,200	LOAD B med 200
B0	OR B	Sammenlign 100 og 200 (A og B)
0600	LD B,0	Nulstil B-registeret
4F	LD C,A	Læg resultatet i C
C9	RET	Returner til BASIC

PRINT USR 30000. Resultatet bliver 236, hvilket fremgår af vor binære udregning herunder.

register A: 01100100 = 100
register B 11001000 = 200
OR B: 11101100 = 236

Som tommelfingerregel kan vi denne gang sige, at resultatet ikke bliver mindre end den største værdi af de to sammenlignede.

Her er alle OR-instruktionerne:

Heks-kode	Mnemonic	
B7	OR A	
B0	OR B	
B1	OR C	
B2	OR D	
B3	OR E	
B4	OR H	
B5	OR L	
B6	OR (HL)	
DDB6	OR (IX+d)	
FDB6	OR (IY+d)	
F6	OR n	hvor d er et tal mellem −128 og 127

OR påvirker flagene på samme måde som AND.

OR bruges ofte til at sætte enkelte bits i A-registeret. For eksempel vil OR 00000111b (OR 7) sætte bit 0, 1 og 2.

XOR-instruktionen er den tredje og sidste logiske instruktion. Sagt kort betyder AND »BÅDE OG«, OR »ENTEN ELLER«, og XOR betyder »DEN ENE, MEN IKKE DEN ANDEN«, hvilket også fremgår af sandhedstabellen:

0 og 0 sammenlignes – resultatet er 0
0 og 1 sammenlignes – resultatet er 1
1 og 0 sammenlignes – resultatet er 1
1 og 1 sammenlignes – resultatet er 0

Som det ses, bliver resultatet kun 1, hvis den ene bit er 0 og den anden 1. Eksempel:

register A: 11010101 = 213
register B: 01110001 = 113
XOR B: 10100100 = 164

Og indsat i en større helhed.

Heks-kode	Mnemonic	Forklaring
3E38	LD A,56	A-registeret LOAD'es med 56
213800	LD HL,56	HL-registeret LOAD'es også med 56
AE	XOR (HL)	Sammenlign A-registeret med værdien på adresse HL (56)
0600	LD B,0	Nulstil B-registeret
4F	LD C,A	Kopiér A i C-registeret
C9	RET	Returner til BASIC

PRINT USR 30000 udskriver resultatet af sammenligningen, og det er 53. Betragtet binært:

register A: 00111000 = 56
XOR (HL): 00001101 = 13 (PEEK 56)
register A: 00110101 = 53

Instruktionerne hedder:

Heks-kode	Mnemonic	
AF	XOR A	
A8	XOR B	
A9	XOR C	
AA	XOR D	
AB	XOR E	
AC	XOR H	
AD	XOR L	
AE	XOR (HL)	
DDAE	XOR (IX+d)	
FDAE	XOR (IY+d)	
EE	XOR n	hvor d er et tal mellem −128 og 127

XOR-instruktionen bruges mest til at vende bits; dvs. at give den enkelte bit den modsatte værdi. For eksempel kan vi ændre bit 7 (sign bit) til den modsatte værdi med XOR 10000000b = XOR 128. Som du måske ved, er tegngeneratoren bygget systematisk op. Kender du koden på et tegn, vil koden for det omvendte tegn (hvidt på sort) være koden plus 128.

Eksempel:
CODE »Z« = 63 = 00111111
CODE »inv Z« = 191 = 10111111

Som det ses, er forskellen kun bit 7 (sign bit). Denne udtrykker jo netop 128.

Vi prøver at illustrere det yderligere her:

Heks-kode	Mnemonic	Forklaring
2A0C40	LD HL,(16396)	
(217D40 – LD HL,16509)		
23	INC HL	HL peger på 1. position på skærmen
363F	LD (HL),63	Skriv et Z her
7E	LD A,(HL)	Hent Z's kode op i A-registeret
EE80	XOR 128	Vend bit 7

23	INC HL	HL peger på 2. position på skærmen
77	LD (HL),A	Skriv et omvendt Z her
C9	RET	Returnér til BASIC

Du kalder rutinen med RAND USR 30000, da vi jo ikke skal have skrevet værdien af BC ud. Hvad rutinen gør, kan du finde ud af ved at læse vor forklaring til programmet.

Som du måske har gættet, påvirker XOR flagene på samme måde som AND og OR.

Kapitel 7

I maskinkode skal man også kunne hoppe rundt i programmer. Instruktionen hertil hedder JP (JumP), men en beslægtet instruktion kaldet JR (Jump Relative) er mere nyttig. JP gemmer vi lidt, hvorimod vi vil kigge nærmere på JR nu.

Med en JR-instruktion kan man hoppe et vist antal bytes frem eller tilbage. Mulighederne ligger imellem −128 og 127 bytes, hvilket er i overensstemmelse med det før omtalte 2-komplement-system. Koden for JR er 18h. F.eks. vil JR 100 have heks-koden 1864, mens JR −60 vil være 18C4. −60 angives i heks-kode som 256−60 = 196, og det er C4h.

JR-instruktionen kan gøres betinget, dvs. kan styres af et flag (Z eller C-flaget). En instruktion som JR Z vil kun blive udført, hvis Z-flaget i udførelsesøjeblikket er sat. Her er de mulige JR-instruktioner:

Heks-kode	Mnemonic
18	JR
20	JR NZ
28	JR Z
30	JR NC
38	JR C

Eksempelvis vil JR NC kun blive udført, hvis carry-flaget er nul, dvs. resat!

Når man bruger JR-instruktionen, er det vigtigt at vide, at programpilen altid flyttes, så den peger på instruktionen efter den, der skal udføres, INDEN denne udføres. Programpilen er et register, der fortæller processoren, hvor i adresseområdet den næste instruktion, der skal udføres, står. For eksempel vil programpilen (PC = Program Counter), inden en JR-ordre udføres, altid blive flyttet, så den peger 2 bytes længere frem.

Hvis ZX 81 udfører en JR-funktion, hvor der springes fremad i programmet (0 til 127 bytes), vil de næste bytes svarende til tallet efter JR blive sprunget over. Vi kan illustrere det med en tegning:

```
byte 30000    JR  }
     30001    100 } JR 100 fylder to bytes
     30002
     30003
     ...
     30101
     30102    Der hoppes hertil
     30103
```

JR 0 vil være en nytteløs ordre, da intet springes over.

Hvis ZX 81 udfører en JR-instruktion, hvor der hoppes bagud (−128 til −1 byte(s)), skal man ved tællingen af antal bytes, der skal hoppes, tælle JR-instruktionens to bytes med. Vi betragter endnu en tegning:

```
byte 30000
     30001
     30002    Der hoppes hertil
     30003
     ...
     30009
     30010    JR  }
     30011    −10 } JR −10 fylder to bytes

     30012
```

Hermed hoppes der til byte 30002, da 30012−10 = 30002. Med ordren JR −2 vil man komme i en uendelig løkke.

Vi vender tilbage til brugen af JR-instruktionen. Først skal vi kigge på en afart af den, kaldet DJNZ. DJNZ betyder Decrement Jump Non Zero, og DJNZ udfører faktisk en hel lille rutine:

DEC B
JR NZ,d hvor d er et tal mellem −128 og 127

DJNZ d virker på den måde, at B først decrementeres, hvorefter den springer d bytes frem/tilbage, hvis B-registerets indhold er forskelligt fra nul. Ellers går den blot videre. Hvis B-reg's indhold er nul fra starten, vil løkken løbes igennem 256 gange, fordi B decrementeres før det undersøges om det er nul.

Vi illustrerer DJNZ med dette eksempel:

Heks-kode	Mnemonic	Forklaring
3E00	LD A,0	A-registeret LOAD'es med nul
0664	LD B,100	B-registeret får værdien 100
3C	INC A	A-registeret incrementeres
10FD	DJNZ −3	Decrementer B og hop til INC A
4F	LD C,A	Kopiér A-registerets indhold i C
C9	RET	Returner til BASIC

Skriv PRINT USR 30000.

Resultatet bør være 100, da A incrementeres 100 gange. Bemærk, at LD B,0 er udeladt. B er jo nul efter nedtællingen.

Ingen af JR-instruktionerne eller DJNZ påvirker flagene. DJNZ påvirker altså ikke Z-flaget, til trods for at B-registeret tælles til nul (man skulle tro, at Z-flaget ville blive sat).

Nu er det – på baggrund af de to sidste kapitlers gennemgang – tid til at lave et program, der virkelig viser maskinkodens hastighed og styrke.

Her er programmet – kommentarerne følger derefter:

Heks-kode	Mnemonic	Forklaring
2A0C40	LD HL,(16396)	Udskriftfilen lokaliseres i HL
(217D40 – LD HL,16509)		
1618	LD D,24	LOAD D med 24 (antallet af linier på skærmen)
0620	LD B,32	LOAD B med 32 (antallet af tegn pr. linie)

23	INC HL	HL peger på næste tegn på skærmen
7E	LD A,(HL)	Hvilket tegn står dér?
EE80	XOR 128	Vend det
77	LD (HL),A	Send det tilbage på skærmen
10F9	DJNZ −7	Hop tilbage, hvis linien ikke er fuldendt
23	INC HL	Spring N/L over
15	DEC D	Decrementer D (antallet af linier)
20F3	JR NZ, −13	Er der flere linier, så hop tilbage
C9	RET	Ellers returner til BASIC

Denne rutine vender skærmbilledet. Vi vil lige demonstrere det. Når du har kørt programmet ind via heks-loaderen, indtaster du følgende BASIC-program:

```
200 FOR N=1 TO 50
210 PRINT AT RND*21,RND*31;CHR$(RND*63+(128 AND
    RND>.5))
220 NEXT N
230 RAND USR 30000
```

Efter RUN 200 vil BASIC-programmet skrive 50 tilfældige tegn rundt om på skærmen, og i linie 230 kaldes så rutinen, der vender skærmbilledet.

Lagde du mærke til hastigheden??

Til sammenligning kan vi sige, at i BASIC vil omvendingen, også kaldet inverteringen, ifølge vore bedste forsøg tage 80 sekunder!! Hvis skærmen er blank (hvid), vil den naturligvis blive sort. Bemærk, at de sidste to linier på skærmen (der normalt bruges til redigering) også vendes (eller inverteres) (D LOADES med 24 linier). I BASIC kan man jo også POKE på de nederste linier. Vi vil senere vende tilbage til netop disse to linier.

Da DJNZ-instruktionen opererer på B-registeret, som jo er et 8 bits-register, er det klart, at man højst kan få 256 gennemløb, netop når B er nul. Men ønsker du flere gennemløb kan du bruge denne rutine:

```
LD DE,60000
opgave
....
DEC DE
LD A,E
OR D
JR NZ,opgave
```

Opgaven kan være en hvilken som helst, der blot skal gennemløbes et antal gange, her 60000. Hvis der ikke er nogen opgave, vil der blot opstå en tom løkke, og der dannes derfor en pause, hvilket vi vender tilbage til.

Rutinen sammenligner bittene i D og E ved en OR-instruktion. Vi husker, at OR giver 1, hvis én af de to sammenlignede bits er 1. Derved opnås et 0-resultat i register A, hvis DE er talt ned til 0, og Z-flaget sættes. Hvis DE er forskellig fra nul, vil Z-flaget ikke blive sat, og JR NZ,opgave hopper tilbage.

Hvorfor vil DEC DE ikke sætte Z-flaget? Det skyldes, at Z-flaget kun sættes ved 8 bits-DEC-instruktioner, og vi behøver derfor OR-instruktionen.

Vi vil nu gå tilbage til de omtalte JP-instruktioner. Dog skal du lige vide, at du allerede nu har benyttet ca. 35% af instruktionerne, og at det er nogle af de vigtigste, som du har lært.

JP-instruktionen hopper ikke relativt i forhold til ordrens placering i arbejdslageret, men til den adresse, der angives efter instruktionen. Eksempel: JP 30000 (heks-kode C33075) vil medføre hop til byte nr. 30000.

JP-instruktionen kan også være afhængig af flagene, og det endda af de fire mest benyttede, nemlig C, Z, S og P/V-flaget.

Instruktionerne er:

Heks-kode	Mnemonic	
C2	JP NZ, nn	Z ikke sat, nn er et tal mellem 0 og 65535
CA	JP Z, nn	Z sat
D2	JP NC, nn	C ikke sat
DA	JP C, nn	C sat
E2	JP PO, nn	P/V ikke sat
EA	JP PE, nn	P/V sat
F2	JP P, nn	S ikke sat
FA	JP M, nn	S sat
C3	JP nn	
E9	JP (HL)	
DDE9	JP (IX)	
FDE9	JP (IY)	

JP NZ, JP Z, JP NC og JP C fungerer næsten som JR-instruktionen. Instruktionen JP NZ,30000 bliver udført, hvis Z-flaget er nul, men der hoppes ikke 30000 bytes frem, men derimod til adresse 30000!

JP PO betyder JumP Parity Odd, og hermed menes, at instruktionen udføres, hvis P/V=0. Tilsvarende virker JumP Parity Even, og springet foretages, hvis P/V-flaget er sat.

JP P betyder JumP Plus. Hoppet udføres, såfremt S-flaget er nul. S-flaget er jo en kopi af sign bit, bit 7, der er nul, hvis tallet er positivt i 2-komplement-metoden. JP M (JumP Minus), hvis S-flaget er sat.

Ønsker du at hoppe p.g.a. overflow, bruges naturligvis JP PO/JP PE, da overflow-flaget jo er P/V-flaget.

JP (HL) bevirker, at der hoppes til HL-registerets adresse. Hvis HL indeholder værdien 30000, hoppes der til adresse 30000 og *ikke* til PEEK HL+256*PEEK (HL+1). Tilsvarende virker JP (IX) og JP (IY). JP berører ikke flagene.

Det er lidt svært umiddelbart at bruge JP-instruktionen, da JR vil være at foretrække til vore små rutiner. Imidlertid vil vi gerne illustrere instruktionen, og vi kan derfor ændre vores skærmvendings-program til dette:

Adresse	Heks-kode	Mnemonic
30000	2A0C40	LD HL,(16396)
	(217D40 – LD HL, 16509)	
30003	1618	LD D,24
30005	0620	LD B,32
30007	23	INC HL
30008	7E	LD A,(HL)
30009	EE80	XOR 128

30011	77	LD (HL),A
30012	05	DEC B
30013	C23775	JP NZ,30007
30016	23	INC HL
30017	15	DEC D
30018	C23575	JP NZ,30005
30021	C9	RET
30022		

RAND USR 30000 og rutinen skulle virke akkurat som den før nævnte. Det fungerer på samme måde, men det fylder lidt mere og optager dermed unødvendig plads. Til gengæld har JR-instruktionen indskrænkninger på 128 bytes tilbage eller 127 frem, og den kan kun gøres betinget af 2 af flagene.

Kapitel 8

I maskinkode har man også brug for at sammenligne registre. Til dette formål bruges instruktionen CP (ComPare), der sammenligner A-registerets indhold med enten A, B, C, D, E, H eller L-registerets indhold eller (HL), (IX+d), hvor d er et tal mellem −128 og 127, men sammenligningen kan også være med et tal n mellem 0 og 255. Instruktionerne er:

Heks-kode	**Mnemonic**
BF	CP A
B8	CP B
B9	CP C
BA	CP D
BB	CP E
BC	CP H
BD	CP L
BE	CP (HL)
FE	CP n
DDBE	CP (IX+d)
FDBE	CP (IY+d)

Disse instruktioner sætter flagene på følgende måde: hvis de to sammenlignede værdier er lig hinanden sættes Z-flaget, og C-flaget resættes. Hvis A-registerets indhold er større end den sammenlignede værdi, resættes både Z- og C-flaget. Hvis A-registerets indhold er mindre end den sammenlignede værdi resættes Z-flaget, og C-flaget sættes.
A-registerets indhold ændres ikke efter instruktionen, kun flagenes status vil skifte. CP A sammenligner A med A. Da disse er ens – naturligvis – vil Z-flaget sættes og C-flaget resættes. Dette er altså en metode til at få Z-flaget sat uden påvirkning af registre, hvis man – mod forventning – behøver det.

Inden vi bringer et programeksempel vil vi gøre rede for en anden instruktion. Den hedder

```
EX DE,HL
```

og denne ombytter indholdet i DE- og HL-registrene. Der findes flere *EXchange*-instruktioner, men dem vil vi vende tilbage til senere.

Som eksempel på brugen af CP har vi udarbejdet en version af det klassiske RENUMBER-program. Linierne i et BASIC-program vil blive omnummereret. Hermed menes, at der bliver et fast interval, f.eks. 10, mellem hvert linienummer. Inden vi går i gang, skal du lige have at vide, hvorledes programfilen er opbygget.

De første to bytes indeholder linienummer. Den ene byte, den første, er den såkaldte MSB, der tæller antallet af 256'ere. Den anden byte, LSB, tæller enere, og for at forvirre dig komplet skal det nævnes, at dette system er lige modsat det normale. For eksempel vil linie 2000 være 256 gange 7 plus 208 i dette system, og den første byte, MSB, får indholdet 7, den anden byte, LSB, indholdet 208. De to næste bytes fortæller, hvor lang linien er. De 4 bytes indeholdende linienummer og længde tælles ikke med, men derimod medregnes det afsluttende Newline. Den første af de to bytes er LSB, der tæller enere, og den anden – egentlig den 4. i hver linies opbygning – tæller 256'ere.

Efter programfilen kommer, på ZX 81, udskriftfilen, som starter med et Newline, værdi 118, se kapitel 1. På Lambda vil programfilen være efterfulgt af 255. Når den sidste linie i BASIC-programmet er forbi, vil der derfor være 2 118-værdier (118 efterfulgt af 255 på Lambda), nemlig enden på den sidste BASIC-linie og starten på udskriftfilen.

Vi vil nu gennemgå RENUMBER-programmet i detaljer.

Vi behøver en del tællere. Vi skal bruge én, der fortæller på hvilken adresse i programfilen vi er. Til dette brug benytter vi HL-registeret, da vi skal både POKE'e og PEEK'e, og til dette formål er det bedst at bruge HL-registeret. Da programfilen starter på adresse 16509 (17302), vil vi skrive


```
LD HL,16509
(LD HL,17302)
```

Vi vil også bruge en tæller, der holder rede på, hvilket linienummer, vi skal bruge. 1. linie skal hedde linie 10, 2. linie linie 20 osv. Vi benytter DE-registeret til dette formål. Ved starten har det værdien 0:

```
LD DE,0
```

Intervallet på 10 skal lægges til DE-registeret ved hver linie, og da 16 bits-addition kun kan ske gennem to registre, bruger vi BC-registeret til at indeholde intervalværdien 10. Denne værdi skal lægges til hver gang, og derfor vil vi vende tilbage til netop dette sted efter hver linie:

```
LD BC,10
```

Nu skal vi have lagt 10 til DE-registeret, så linienummerintervallet holdes efter hver linie. Da vi ikke har 16 bits-addition med formen ADD DE,BC må vi først ombytte HL og DE:

```
EX DE,HL
```

Så kan vi lægge 10 til:

```
ADD HL,BC
```

Vi skal nu normalisere forholdene, og derfor ombytter vi igen:

```
EX DE,HL
```

Så har vi igen linienummertælleren i DE, og programfilens adresse i HL. På den første adresse, der er 16509 (17302) ved 1. gennemløb, ligger MSB af linienummeret. Ved omnummereringen ligger MSB af linienummeret i D-registeret. Derfor skal vi overføre det omnummererede linienummers MSB (D-registeret)

til byte 16509 (17302). HL peger allerede på denne byte. Vi bruger instruktionen

```
LD (HL),D
```

Vi skal nu arbejde med LSB af linienummeret, der ved 1. gennemløb er gemt i 16510 (17303). Vi incrementerer HL.

```
INC HL
```

Det omnummererede linienummer's LSB er gemt i E-registeret, og dette erstatter nu det oprindelige linienummers LSB:

```
LD (HL),E
```

Ved 1. gennemløb, hvor DE indeholder værdien 10, vil D være 0 (MSB af 10) og E 10 (LSB).

På de næste adresser ligger længden af linien minus de fire første bytes. Dette tal skal vi bruge for at finde enden på linien, så vi incrementerer HL, så den peger på næste adresse.

```
INC HL
```

Ved 1. gennemløb peger HL nu på 16511 (17304). Vi skal lige huske, at her ligger LSB først. Vi benytter BC-registeret til at gemme længden i, ihukommende, at vi giver BC intervalværdien 10, når vi hopper til starten af rutinen. LSB ligger i C-registeret, så vi skriver

```
LD C,(HL)
```

Naturligvis skal MSB, i 16512 (17305) ved 1. gennemløb, ned i B-registeret, så vi incrementerer HL:

```
INC HL
```

og bruger instruktionen


```
LD B,(HL)
```

Derefter lægger vi BC til HL, da længden af linien, nu gemt i BC, skal lægges til den tæller, der peger på adresserne i programfilen. Vi kan umiddelbart klare opgaven med

```
ADD HL,BC
```

Nu peger HL på enden af linien: Newline med koden 118. Byten efter denne vil indeholde MSB af den næste linie, medmindre der ikke er flere! Da indeholder den jo 118, der er starten på udskriftfilen på ZX 81. (På Lambda vil den indeholde 255). Vi incrementerer HL, så den nu peger på denne byte:

```
INC HL
```

Hvis denne byte har indholdet 118 (255), skal der returneres til BASIC. Her bruger vi netop CP-ordren. Vi husker imidlertid, at CP-instruktionen kun virker ved sammenligning mellem A-registeret og et andet register. Derfor lægger vi først værdien 118 (255) op i A-registeret.

```
LD A,118
(LD A,255)
```

Nu sammenligner vi dette med indholdet af den omtalte byte – og den peger HL jo på.

```
CP (HL).
```

Hvis koden på adresse HL er 118 (255) – dermed vil programfilen være slut – vil Z-flaget blive sat. Derfor skal der returneres til BASIC, hvis Z er sat. Sagt på en anden måde skal rutinen gennemløbes igen, hvis Z-flaget *ikke* er sat. Med

```
JR NZ,–20
```

hopper vi tilbage, og rutinen gennemløbes atter. At der netop skal hoppes 20 tilbage, kan du forvisse dig om ved at kigge på det færdige program herunder.

Endelig skal der naturligvis returneres, hvis Z-flaget er sat, og der ikke hoppes tilbage:

```
RET
```

Programmet bliver i sin helhed således:

Heks-kode	Mnemonic
217D40	LD HL,16509
(219643 – LD HL,17302)	
110000	LD DE,00
010A00	LD BC,10
EB	EX DE,HL
09	ADD HL,BC
EB	EX DE,HL
72	LD (HL),D
23	INC HL
73	LD (HL),E
23	INC HL
4E	LD C,(HL)
23	INC HL
46	LD B,(HL)
09	ADD HL,BC
23	INC HL
3E76	LD A,118
(3EFF – LD A,255)	
BE	CP (HL)
20EC	JR NZ,–20
C9	RET

Hvis du ønsker at omrenummerere med 100, ændrer du bare LD BC,10 til LD BC, 100.

Kapitel 9

I dette kapitel vil vi gennemgå nogle meget simple og forståelige kommandoer. Disse instruktioner er der ikke meget at fortælle om.

Den første hedder CPL (heks-kode 2F). Denne instruktion virker kun på A-registeret. Den vender simpelt hen alle 8 bit i A.

Hvis A er 10011010b vil registeret efter CPL være 01100101b.

Den næste hedder NEG (ED44 i heks-kode). Denne opererer også kun på A. Den finder 2-komplementet, som før omtalt, og det er altså 256 minus A-registerets indhold før instruktionen. Med andre ord vendes hver enkelt bit og der adderes 1. Hvis A er 10010101b (149), vil registeret efter NEG være 01101011b (107).

Instruktionen med heks-kode 37 hedder SCF, og den sætter C-flaget.

I familie med den er CCF (med heks-kode 3F). Den vender C-flagets indhold. Er flaget sat, resættes det med CCF, og modsat naturligvis.

NOP (No OPeration) har heks-koden 00. Denne instruktion gør – som navnet fortæller – ingenting. Hvis man ønsker at slette noget i et maskinkodeprogram, kan man blot overstrege det med instruktionen NOP, der passende har kode 0, som er mellemrums-karakterkoden.

Det var de mest simple.

Nu introducerer vi en anden instruktion, og den kan optræde i mange former. Den hedder SET X,r. Disse instruktioner sætter bit X, hvor X er et tal mellem nul og syv, i ét af følgende registre: A, B, C, D, E, H eller L, (HL), (IX+d) eller (IY+d). Med (HL) menes som sædvanlig, at der sættes en bit X i værdien på byte HL. Vi vil vise dette:

Heks-kode	Mnemonic	Forklaring
2A0C40	LD HL,(16396)	Lad HL pege på udskriftfilen
(217D40 – LD HL,16509)		
23	INC HL	HL peger på 1. position på skærmen
3614	LD (HL),20	Skriv et "=" dér (koden er 20)
11FFFF	LD DE,65535	Tom løkke, nødvendig for at kunne se, hvad der sker
1B	DEC DE	
7B	LD A,E	
B2	OR D	
20FB	JR NZ, –5	
CBFE	SET 7,(HL)	HL peger på det sted i udskriftfilen, hvor vi har sat =. Ved koden 20 er bit 7 ikke sat, men ved at gøre det, fås den omvendte gengivelse af tegnet.
C9	RET	Returner til BASIC

Læg i øvrigt mærke til, hvor hurtigt en løkke på 65535 klares. Det tager ca. 2 sekunder, hvor det tager omkring 20 minutter i BASIC i SLOW.

Hvis bitten er sat inden SET-instruktionen, sker der ingenting.

RES X,r er den modsatte instruktion. Mulighederne er de samme som med SET, men nu resættes bit X i stedet for at blive sat.

Prøv selv i et lignende program som det ovenfor, men erstat LD (HL),20 med LD (HL),148 og SET 7,(HL) med RES 7,(HL).

BIT X,r arbejder også med A, B,...,(IY+d). Denne instruktion finder ud af, om bit X er sat i et af ovenstående registre. Hvis bit X er sat (=1) vil Z-flaget blive resat. Hvis bitten er resat (=0) vil Z-flaget blive sat. Z-flaget tager altså den omvendte værdi af den testede bit. Hvis A er 10011010b og BIT 3,A bliver udført, vil Z-flaget blive 0, da bit 3 i A (den 4. bit fra højre) er 1.

Alle heks-koderne for SET, RES og BIT vil vi ikke skrive op. Der er 240 af dem sammenlagt, og du kan se dem i appendiks A.

Vi vil derimod lave et program, der udskriver alle displaybare tegn, dvs. alle tegn, som kun fylder en plads på skærmen – i modsætning til »PRINT« f.eks. Alle disse displaybare tegn har resat bit 6, og det betyder, at vi kan bruge BIT-instruktionen.

Heks-kode	Mnemonic	Forklaring
2A0C40	LD HL,(16396)	Identificer udskriftfilen
(217D40 – LD HL,16509)		
23	INC HL	HL peger på 1. position på skærmen
0600	LD B,0	Giv B-registeret værdien nul
04	INC B	Lad B være næste tegn
3EFF	LD A,255	Giv A-registeret indholdet 255
B8	CP B	Sammenlign med B
C8	RET Z	Returner, hvis B er 255
CB70	BIT 6,B	Er tegnet displaybart?
20F7	JR NZ, –9	Hvis ikke, hop til INC B
70	LD (HL),B	Skriv det
111027	LD DE,10000	Pause
1B	DEC DE	
7B	LD A,E	
B2	OR D	
20FB	JR NZ, –5	
18EC	JR –20	Hop tilbage til INC B

Vi bliver nødt til at forklare instruktionen RET Z. Denne instruktion returnerer simpelt hen (til BASIC i dette tilfælde), hvis Z-flaget er sat. I programmet bliver Z-flaget sat, hvis B har værdien 255 (ensbetydende med at der ikke er flere tegn), og RET Z vil derfor returnere. Disse RET-instruktioner kommer der mere om senere, hvor vi vil forklare, hvordan de virker.

Kapitel 10

Der findes endnu et register i Z80-assembler-sproget. Dette 16 bits-register hedder stack-pointeren (SP), og det kan ikke opdeles i to 8 bits-registre, hvilket jo også gælder for IX- og IY-registrene.

SP peger på toppen af stakken. Stakken er et sted, hvor man kan gemme værdier. Værdierne lægges oven på hinanden i stakken, og kun den øverste værdi kan tilbagekaldes. Man kan ikke snuppe værdier fra midten af stakken!!! For at forvirre, så ligger bunden af stakken (dvs. dér, hvor den 1. værdi ligger) øverst i RAM. En del af RAM, den såkaldte maskinstak, er forbeholdt stakken, og når der lægges værdier på stakken, vil de altså blive placeret længere og længere nede i RAM.

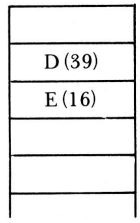
Stakken ligger f.eks. på 32765 (SP peger på denne adresse). Når der lægges en værdi oven på stakken, vil den blive lagt på adresserne 32764 og 32763. Da vil SP formindskes med to, og SP vil indeholde værdien 32763, som nu er toppen af stakken. Som det kan ses, vil værdierne altså blive lagt i nedadgående retning i RAM.

For at lægge værdier op på stakken kan vi bruge én af følgende instruktioner:

Heks-kode	Mnemonic
F5	PUSH AF
C5	PUSH BC
D5	PUSH DE
E5	PUSH HL
DDE5	PUSH IX
FDE5	PUSH IY

PUSH-kommandoen lægger altså værdien af én af ovenstående 16 bits-registre oven på stakken. Eksempel: Hvis SP indeholder

30000, og DE indeholder værdien 10000 (256 x 39 + 16), så vil PUSH DE-instruktionen forløbe således:



30000 ← SP før SP=30000
29999
29998 ← SP efter SP=29998
29997
29996

Som det ses på tegningen, vil SP først blive decrementeret, dernæst vil værdien af det mest betydende register i 16 bits-registeret, her D-registeret, blive lagt på stakken. SP decrementeres igen til 29998 i dette tilfælde, og E, det mindst betydende register, vil blive placeret her. Faktisk udføres denne rutine:

```
DEC SP
LD (SP),D
DEC SP
LD (SP),E
```

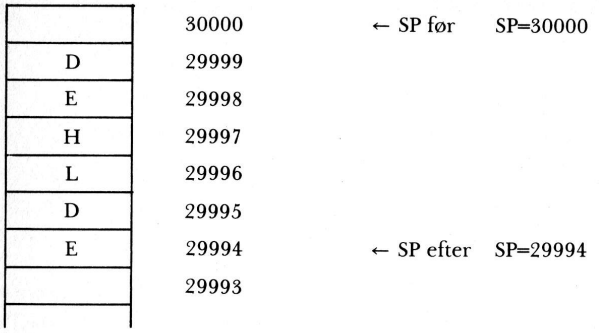
Du bedes bemærke, at LD (SP),D og LD (SP),E ikke findes i Z80-assembler-sproget. De er kun medtaget for at gøre PUSH-instruktionen (her PUSH DE) mere overskuelig. Der findes kun 16 bits-PUSH, men ovenstående forklarer forhåbentlig begreberne. Vi mener, at PUSH BC, PUSH DE, PUSH HL, PUSH IX og PUSH IY giver sig selv, men PUSH AF kræver en nøjere forklaring.

A-registeret er som nævnt et decideret 8 bits-register, men her finder vi det sammen med et andet velkendt 8 bits-register, nemlig flagregisteret, F! Til trods for, at de lægges sammen på stakken, så fungerer de stadig som to 8 bits-registre.

Hvis vi nu i stedet skriver, f.eks.

```
PUSH DE
PUSH HL
PUSH DE
```

vil stakken se således ud:



Tilsvarende skal vi naturligvis have en instruktion, som gør det diametralt modsatte af PUSH. Denne skal lægge værdien fra toppen af stakken tilbage til ét af registrene. Til dette brug har vi POP-instruktionen, der virker på de samme registre som PUSH:

Heks-kode	Mnemonic
F1	POP AF
C1	POP BC
D1	POP DE
E1	POP HL
DDE1	POP IX
FDE 1	POP IY

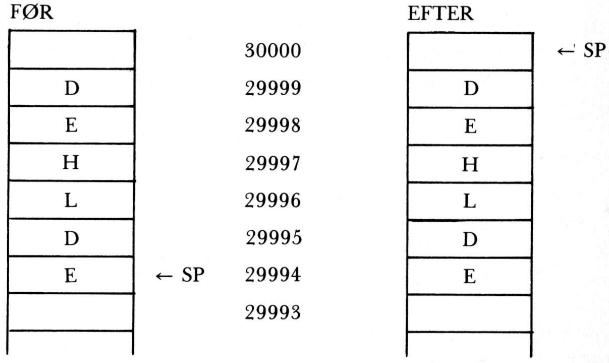
POP DE udfører dette lille »program«:

```
LD E, (SP)
INC SP
LD D, (SP)
INC SP
```

Endnu en gang understreger vi, at instruktionerne LD E, (SP) og LD D, (SP) ikke findes! Der eksisterer også kun 16 bits-POP.

Vi kigger på skemaet fra før og udfører disse instruktioner:


```
POP DE
POP HL
POP DE
```



Bemærk, at de POP'ede værdier stadie er bibeholdt i RAM. SP (stack-pointer) har blot flyttet sig til den nye top af stakken, og tallene er overført til registrene.

Du mindes forhåbentlig stadig, at

```
PRINT USR 30000
```

ved RET-instruktionen vil udskrive BC-registerets indhold. Det bruger vi igen.

Eksempel:

Heks-kode	Mnemonic	Forklaring
111027	LD DE,10000	Læg værdien 10000 i DE-registeret
D5	PUSH DE	Put DE op på stakken
C1	POP BC	Hiv toppen af stakken ned i BC
C9	RET	Returner til BASIC

Skriv PRINT USR 30000. Resultatet bør blive 10000.

Hvis vi ønsker at ombytte DE og BC, kan vi skrive

```
PUSH DE
PUSH BC
POP DE
POP BC
```

Hvis vi ønsker at ombytte D og E uden at berøre andre registre, må vi finde en anden udvej, da der ikke findes EX D,E. Men inden må vi have indført en anden instruktion vedrørende SP, nemlig

```
INC SP
```

Heks-kode	Mnemonic
3B	DEC SP
33	INC SP

Disse virker på samme måde som de øvrige INC og DEC-instruktioner. Her er det blot SP, der peger på toppen af stakken, der ændrer værdi.

Til at ombytte D og E starter vi med at PUSH'e DE to gange. Nu ser stakken således ud:

	← SP før	30000
D		29999
E		29998
D		29997
E	← SP efter	29996

Derefter vil vi incrementere SP, så stakken ser således ud. Læg mærke til, hvor SP peger til.

D		29999
E		29998
D	← SP nu	29997
E		29996

Derefter POP'er vi DE. Som før beskrevet bliver E-registeret LOAD'et med (SP). Herved bliver E altså nu lig D, og derefter incrementeres SP, og D LOAD'es med (SP), der er E i dette tilfælde. Nu har vi byttet om på D og E, og stakken ser således ud:

D	← SP nu	29999
E		29998
D		29997
E		29996

Derefter mangler vi bare at incrementere SP tilbage til udgangspunktet.

Heks-kode	Mnemonic
D5	PUSH DE
D5	PUSH DE
33	INC SP
D1	POP DE
33	INC SP

Vi har et eksempel mere:

Heks-kode	Mnemonic	Forklaring
011027	LD BC,10000	Læg værdien 10000 i BC-reg
C5	PUSH BC	Læg BC på stakken
C5	PUSH BC	Læg igen BC på stakken
33	INC SP	Incrementér stack-pointer
C1	POP BC	Læg værdien på toppen af stakken ned i BC
33	INC SP	Incrementér SP
C9	RET	Returner til BASIC

Skriv PRINT USR 30000. Du vil måske forvente et resultat på 10000, men B og C er blevet ombyttet. Før indeholdt BC 10000, der er $256 \times 39 + 16$, så B var 39 og C var 16. De er nu blevet

ombyttet, så B er 16, og C er 39. Derfor indeholder BC 256 × 16 + 39 = 4135. Dette var resultatet, forhåbentlig!

Der findes andre kommandoer vedrøernde SP:

Heks-kode	Mnemonic	
31	LD SP,nn	hvor nn er et tal mellem 0 og 65535
F9	LD SP,HL	
ED7B	LD SP,(nn)	
ED73	LD (nn),SP	
39	ADD HL,SP	
ED7A	ADC HL,SP	
ED72	SBC HL,SP	
DD39	ADD IX,SP	
FD39	ADD IY,SP	
E3	EX (SP),HL	
DDE3	EX (SP),IX	
FDE3	EX (SP),IY	
FDE3	**EX (SP),IY**	

LD SP,nn LOAD'er SP med et tal mellem 0 og 65535, og herved kan man starte en ny stak.

LD SP,HL kopierer HL's indhold i SP.

LD SP,(nn) LOAD'er SP med et 16 bits-tal, der findes med

PEEK (nn)+256*PEEK (nn+1)

LD (nn),SP giver værdien af SP til to bytes, nn og nn+1. Den bliver på to adresser, da SP jo er et 16 bits-register.

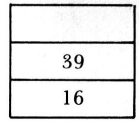
ADD HL,SP adderer stack-pointeren til HL-registerets indhold, og hedder instruktionen ADC HL,SP tillægges endvidere værdien af carry-flaget.

SBC HL,SP trækker stack-pointeren fra HL-registerets indhold, og yderligere trækkes en eventuel carry fra.

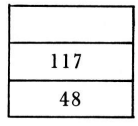
ADD IX,SP og ADD IY,SP adderer stack-pointeren til indholdet af hhv. IX- og IY-registeret.

EX(SP),HL ombytter toppen af stakken med HL.

Eksempel: HL er 30000, der er 256 x 117 + 48, og 10000 (256 x 39 + 16) er blevet PUSH'et op på stakken via et register.



Efter EX (SP),HL ser det således ud:



SP-registeret har samme værdi. Stack-pointeren er ikke ændret.

På samme måde virker EX (SP),IX og EX (SP),IY.

Der findes ingen EX (SP),BC instruktion, men det kan ordnes med dette lille program:

```
EX (SP),HL
PUSH BC
EX (SP),HL
POP BC
EX (SP),HL
```

Samme fremgangsmåde kan benyttes, hvis man ønsker at ombytte toppen af stakken med DE eller AF. Erstat BC i ovenstående eksempel med det ønskede register.

PUSH og POP berører ikke flagene, men der er en undtagelse, nemlig POP AF. Flagene vil naturligvis antage den værdi, som de får ved POP AF-instruktionen.

Vi vil senere vende tilbage til PUSH og POP og de øvrige stak-instruktioner i praktisk anvendelse.

Kapitel 11

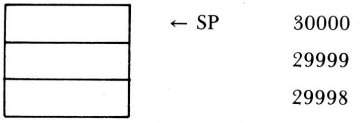
I BASIC findes kommandoen GOSUB. I maskinkode findes instruktionen CALL, der ligesom GOSUB kalder en underrutine. CALL kan være afhængig af flagene C, S, Z og P/V ligesom JP-instruktionen.

Heks-kode	Mnemonic	
CD	CALL nn	hvor nn er et tal mellem 0 og 65535
C4	CALL NZ,nn	
CC	CALL Z,nn	
D4	CALL NC,nn	
DC	CALL C,nn	
E4	CALL PO,nn	
EC	CALL PE,nn	
F4	CALL P,nn	
FC	CALL M,nn	

Ved CALL nn kaldes en rutine, som begynder på adresse nn, uanset flagenes status. CALL NZ udføres, udførehvis Z-flaget er 0, CALL Z udføres, hvis Z-flaget er 1. Her er en fuldstandig liste:

CALL NZ udføres, hvis Z-flaget er resat (nul)
CALL Z udføres, hvis Z-flaget er sat (en)
CALL NC udføres, hvis C-flaget er resat (nul)
CALL C udføres, hvis C-flaget er sat (en)
CALL PO udføres, hvis P/V-flaget er resat (nul)
CALL PE udføres, hvis P/V-flaget er sat (en)
CALL P udføres, hvis S-flaget er resat (nul)
CALL M udføres, hvis S-flaget er sat (en)

Ligesom GOSUB husker CALL også en returadresse. Denne lagres på den måde, at den PUSH'es op på stakken. Hvis stakken til at begynde med ser således ud:

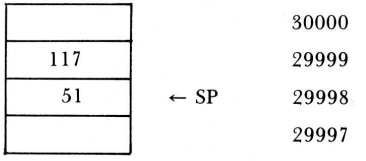


og vi kører dette lille program:

Adresse	Heks-kode	Mnemonic
30000	CD1879	CALL 31000
30003		...

så er det ikke de 31000, der bliver PUSH'et op på stakken, men derimod adressen efter CALL-instruktionen.

I dette tilfælde bliver tallet 30003, der er første instruktion efter CALL, lagt op på stakken. 30003 er 256 x 117 + 51, så stakken vil nu se således ud:



CALL kalder altså ligesom GOSUB et underprogram, en såkaldt subrutine. Subrutiner bruges, hvis en sekvens af et program skal udføres flere gange. Da kan subrutinen kaldes igen og igen. Vi tager et eksempel på brugen af CALL, uden at vi dog anvender kommandoen særlig smart. Til illustration hiver vi vores tegn program fra kapitel 9 frem igen.

Når du udfører programmet, skal du trykke BREAK *inden* skærmen bliver helt fuld, da du ellers kan opleve enhver programmørs rædsel: crash'et!!!

Heks-kode	Mnemonic	Adresse	Forklaring
2A0C40	LD HL,(16396)	30000	Udskriftfilen findes
(217D40 – LD HL,16509)			
0600	LD B,0	30003	Tæller nulstilles
LØKKE			
23	INC HL	30005	Næste tegn på skærmen
3E76	LD A,118	30006	LOAD A med 118 (N/L)
BE	CP (HL)	30008	Er det Newline?
28FA	JR Z, LØKKE	30009	så peg på næste tegn
CD4475	CALL PRINTCHR	30011	Tegnrutine kaldes
CD460F	CALL 3910	30014	Er BREAK nedtrykket?
(CD3B11 – CALL 4411)			
D0	RET NC	30017	så returner til BASIC
18F1	JR LØKKE	30018	Hop til LØKKE
PRINTCHR			
04	INC B	30020	Næste tegn
CB70	BIT 6,B	30021	Er det displaybart?
20FB	JR NZ, PRINTCHR	30023	Er det ikke, hop til næste tegn
11D007	LD DE,2000	30025	
1B	DEC DE	30028	
7B	LD A,E	30029	Pause
B2	OR D	30030	
20FB	JR NZ,–5	30031	
70	LD (HL),B	30033	Udskriv tegn
C9	RET	30034	Returner til hovedprogram

Allerførst bør du bemærke, at vi har indført et par nye skrivemåder. Vi har brugt to små rutiner. Den første rutine har vi kaldt LØKKE, og det står ved starten på den. Når vi hopper til den, i byte 30018, skrives JR LØKKE, og der hoppes tilbage til det anførte sted. Samme system gælder ved CALL, og det står anført ved byte 30011. Vi indfører den skrivemåde, så vi ikke behøver tælle frem og tilbage hele tiden. Det er unægteligt noget nemmere at gå op til den angivne etiket, som den slags kaldes, men når der skal oversættes til heksadecimale tal, skal man selvfølgelig tælle. Man kan jo ikke bare skrive LØKKE! Til højre for de anførte instruktioner står nummeret på den byte, hvor instruktionen begynder.

Vi vil nu forklare, hvordan programmet fungerer. Hvis du allerede har forstået det, kan du bare springe forklaringen over.

Hovedprogrammet består af

```
LD HL,(16396)
  (LD HL,16509)
LD B,0
INC HL
LD A,118
CP (HL)
JR Z,–6
CALL PRINTCHR
CALL 3910
  (CALL 4411)
RET NC
JR LØKKE
```

Først finder vi udskriftfilen, hvis startadresse lægges i HL. B-registeret, der holder styr på det næste tegn, vi måske skal POKE ind på skærmen, nulstilles. HL incrementeres, så den peger på det næste tegn på skærmen. Ved første gennemløb er det det første tegn, da udskriftfilen jo starter med N/L. I de næste tre instruktioner kontrollerer vi, om vi næste gang vil POKE på en »end of line«-kode, N/L, og det må vi jo ikke, da detvil forårsage et crash!

Hvis vi vil POKE på et N/L, vil Z-flaget ved sammenligningen blive sat, og vi må atter incrementere HL til næste skærmposition. Dernæst gør vi brug af CALL. Vi kalder vort underprogram, som forklares om lidt. Den næste instruktion er også en CALL, men den kalder en rutine i ROM, og hermed menes én

af de af Sinclair fastlagte rutiner, som ZX 81 selv bruger. Denne undersøger, om BREAK-tasten er nedtrykket. Hvis dette er tilfældet, vil rutinen resætte C-flaget, og der vil blive returneret ved RET NC til BASIC! Hvis BREAK ikke er aktiveret, vil JR hoppe tilbage til starten af løkken, hvor HL incrementeres.

Underprogrammet består af

```
INC B
BIT 6,B
JR NZ,LØKKE
LD DE,2000
DEC DE
LD A,E
OR D
JR NZ,-5
LD (HL),B
RET
```

Incrementer B sørger for, at vi nu kigger på det næste tegn i tegnsættet. Med BIT 6,B undersøger vi, om den sjette bit i B er sat. Du husker vel, at hvis den ikke er sat, så er tegnet displaybart. Hvis den er sat, altså ikke-displaybart, vil Z-flaget resættes, og hvis dette er tilfældet, hoppes tilbage til INC B, og vi undersøger det næste tegn. Hvis tegnet er displaybar, vil programmet fortsætte med resten af subrutinen. Først laver vi en pause – som du jo kender – for at skærmen ikke skal fyldes så hurtigt. Dernæst skrives det displaybare tegn på næste skærmposition. Sluttelig returnerer vi fra underprogram til hovedprogram, naturligvis tilbage til den instruktion, der kommer efter CALL-ordren. Næste instruktion, der udføres, efter RET er derfor

```
CALL 3910 (4411)
```

Programmet er – som tidligere beskrevet – en dårlig måde at bruge CALL på, da underprogrammet blot kunne være placeret i slutningen af hovedprogrammet, men bedre eksempler følger.

Prøv selv at eksperimentere videre med din nuværende viden om instruktionerne.

*

Som du måske har bemærket, har vi brugt kommandoen RET til at vende tilbage fra en maskinkoderutine til BASIC. RETurn returnerer til instruktionen efter det sted, hvor der blev kaldt fra. Returadressen, som stedet hedder, er – som før nævnt – gemt i stakken. I vort netop forklarede tegnprogram brugte vi RET til at vende tilbage fra subrutinen til hovedprogrammet.

I RENUMBER-programmet brugte vi instruktionen RET Z. Her blev der returneret, hvis Z-flaget var sat. Da vi kaldte »BREAK-rutinen« i tegnprogrammet før, blev der returneret på NC, og dermed menes, at C-flaget ikke er sat.

Som du måske har regnet ud, så er RET ligesom CALL en instruktion, der kan afhænge af flagene. Her er de alle ni:

Heks-kode	**Mnemonic**
C9	RET
C0	RET NZ
C8	RET Z
D0	RET NC
D8	RET C
E0	RET PO
E8	RET PE
F0	RET P
F8	RET M

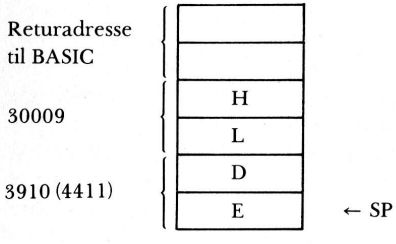
De virker omvendt af CALL. Den POP'er stakken og putter den fundne værdi i PC (program-counter). Som du husker, er PC programpilen, som udpeger den næste instruktion, der skal udføres.

Hvis der er blevet udført en CALL-instruktion, vil returadressen (dér, hvor der skal vendes tilbage til, når subrutinen returnerer), blive lagret på toppen af stakken. Er CALL-ordren brugt i adresse 30000, skal der returneres til 30003, da CALL-instruktionen, som du måske husker, fylder tre bytes, og derfor udfylder

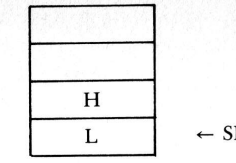
adresserne 30000, 30001 og 30002. Da også PUSH og POP virker på SP, kan man manipulere med returadresserne. RET virker som POP PC. Hvis DE-registeret indeholder 10000, og DE PUSH'es, vil rutinen, når den støder på en RET, sætte PC til 10000 og endvidere incrementere SP to gange som med sædvanlig POP, og eksekveringen fortsætter fra 10000. Et eksempel på manipulation vises nedenfor:

Heks-kode	Mnemonic	Adresse
START		
213975	LD HL,30009	30000
11460F	LD DE,3910	30003
(113B11 – LD DE,4411)		
E5	PUSH HL	30006
D5	PUSH DE	30007
C9	RET	30008
D0	RET NC	30009
18F4	JR START	30010

Vi starter med at LOAD'e HL med værdien 30009. Dernæst LOAD'er vi DE med værdien 3910 (4411). Som du måske erindrer, så ligger BREAK-rutinen i ROM netop på denne adresse. Derefter lægger vi værdien 30009, HL's indhold, op på stakken, og DE's værdi, startadressen på BREAK-rutinen, følger efter. Nu ser stakken således ud:



Nu udføres RET-instruktionen, og der vil blive returneret til den adresse, som står øverst på stakken. I dette tilfælde BREAK-rutinens startadresse. Nu ser stakken således ud:



Denne rutine virker som før beskrevet på den måde, at C-flaget resættes, hvis BREAK er nedtrykket, ellers er C sat. Efter dette støder den på en RET nede i ROM! Denne RET POP'er næste returadresse i stakken. Denne adresse er 30009 (HL). Dengang vi skrev RET i *vores* program, stod næste instruktion netop på 30009. Derfor returnerer ROM-rutinen til 30009. På 30009 står RET NC. Denne returnerer til næste adresse i stakken, nemlig returadressen til BASIC, men vel at mærke kun, hvis BREAK er nedtrykket. Hvis den ikke er det, fortsættes med næste instruktion, der er JR START. Den hopper tilbage til starten af programmet. Det vil sige, at den kører i en løkke, som kun kan brydes af BREAK.

Vi har lovet at bringe et program, der viser forskellige mønstre på skærmen. Det bringer vi her. Det virker kun – med effekt – på ZX 81.

Heks-kode	Mnemonic	Forklaring
3EFE	LD A,254	A LOAD'es med 254
START		
F5	PUSH AF	Værdien lægges på stakken
ED47	LD I,A	Interruptregisteret får denne værdi
111027	LD DE,10000	
1B	DEC DE	
7B	LD A,E	Pause
B2	OR D	
20FB	JR NZ, –5	
F1	POP AF	Værdien hentes tilbage
3C	INC A	Den forøges
FE1F	CP 31	Er den 31? Hvis ja,
C8	RET Z	så returner til BASIC
18EE	JR START	ellers hop til START

I næste kapitel skal vi se nærmere på nogle helt specielle CALL-instruktioner.

Kapitel 12

Der findes en anden slags CALL-instruktioner, hvor subrutine-adressen er fastlagt på forhånd af Zilog. Disse instruktioner hedder RST, og de fylder kun én byte. Der er 8 forskellige RST-instruktioner, og de er som følger:

Heks-kode	Mnemonic
C7	RST 0h
CF	RST 8h
D7	RST 10h
DF	RST 18h
E7	RST 20h
EF	RST 28h
F7	RST 30h
FF	RST 38h

RST-instruktioner kan altså kalde 8 forskellige indbyggede rutiner i ROM med adressen angivet efter RST. De bruges meget flittigt af ZX 81, når den arbejder, og derfor er de placeret netop her, så de kan kaldes med RST. De kan dog kaldes på normal måde med CALL 0h, CALL 8h...,men CALL-instruktionen fylder jo tre bytes.

RST-instruktionen kan ikke betinges af et flags status, til det kan man bruge CALL i stedet.

Vi vil kun arbejde med 4 af de 8 ordrer, da de 4 andre ikke er nyttige for maskinkodeprogrammører, i hvert fald ikke begyndere. Vi vil dog forklare, hvad de står for.

Den første er RST 0h, og den kalder altså adresse 0 i ROM. Denne rutine har næsten samme virkning, som hvis du tager ledningen ud af ZX 81 og sætter den i igen. Den skal man altså holde sig langt fra, hvis man ikke vil miste sine programmer. Hvis du skriver NEW i BASIC, vil du slette alt op til RAMTOP, men RST 0h, eller RAND USR 0 i BASIC om du vil, vil flytte

RAMTOP til den højest mulige position, der er 32768, såfremt du har 16K RAM. Herefter vil den eliminere alt inden for dette område. Har du f.eks. 32K RAM, vil du dog stadig kunne bevare de sidste 16 K, da ZX 81 – som du ved – ikke selv kan finde ud af, at du har 32 K RAM.

På Lambda vil RAND USR 0 (eller RST 0h i et maskinkodeprogram) svare til, at du trykker på RESET-knappen. Ingen programmer forsvinder altså herved.

RST 8h-rutinen er fejlrapporteringsrutinen. Det er denne, som giver dig de små meddelelser, som står nede i venstre hjørne. Denne rutine vender ikke tilbage til det sted, hvorfra der blev kaldt. Den skriver fejlrapporten, og så er man tilbage i BASIC, som havde man været der hele tiden.

Denne kan man bruge i egne rutiner. Det gøres på følgende måde på ZX 81: Efter den byte, hvorpå RST 8h er placeret, skal ligge en kode for den rapport, man ønsker. Koden for disse rapporter er fejlens kode minus 29. Hvis du ønsker et A som rapport skal RST 8h efterfølges af CODE »A«–29, 38–29, hvilket er 9, og programmet vil se således ud:

Heks-kode	Mnemonic
CF	RST 8h
09	DEFB

DEFB betyder DEFine Byte, og hermed menes, at vi bruger en byte til at udtrykke noget, som ikke skal tolkes som en instruktion. Husk for fremtiden, at DEFB ikke er en instruktion, men blot udtrykker, at denne byte er forbeholdt programmet til et bestemt formål. I ovenstående program bruges DEFB, fordi RST 8h skal efterfølges af en bestemt kode.

Du kan også få udskrevet andre fejlkoder end dem, der er standard i ZX 81 BASIC. Prøv f.eks. dette eksempel:

Heks-kode	Mnemonic
CF	RST 8h
22	DEFB

Skriv RAND USR 30000, forudsat rutinen er anbragt derfra, naturligvis. Prøv også

Heks-kode	Mnemonic
CF	RST 8h
81	DEFB

Dette vil give omvendt 2-tal som rapport. Det er muligt at få alle tal og bogstaver i normal og omvendt gengivelse. Prøv dette:

Heks-kode	Mnemonic
START	
CD460F	CALL 3910
(CD3B11 – CALL 4411)	
38FB	JR C, START
CF	RST 8h
0C	DEFB

Hvis BREAK ikke er nedtrykket, er C-flaget én, og der startes forfra. Hvis BREAK aktiveres, kommer fejlkode D, da 0Ch er 12 i decimal notation, og 12 + 29 er 41, der er koden for D.

På Lambda fungerer det lidt på samme måde. Hvis du kigger i instruktionsbogen, kan du se opført 16 forskellige fejlrapporter. OK svarer til 0 i ZX 81-fejlmeddelelse, NF til 1..., AG til A op til MF, der er F. Systemet er derfor det samme. Bruges BREAK-programmet ovenfor, vil rapportkoden være BK svarende til D på ZX 81.

RST 10h er en »skriv et tegn-rutine«. Den udskriver en tegn svarende til indholdet af A-registeret. Hvis A-registeret indeholder værdien 128, vil en RST 10h udskrive en sort firkant, som jo har kode 128, på den næste printposition. Prøv følgende:

Heks-kode	Mnemonic
3E07	LD A,7
START	
D7	RST 10h
18FD	JR START

Skriv RAND USR 30000. Læg mærke til, at vi ikke tager forbehold for koden 118 i hver linie. Det ordner RST 10h selv. Når du kører programmet, bliver skærmen fyldt af det grafiske tegn svarende til kode 7, og det hele ender med fejlkode 5 (SF), der betyder fyldt skærm. RST 10h finder altså ud af, om skærmen er fyldt. Prøv nu:

Heks-kode	Mnemonic
START	
3E07	LD A,7
D7	RST 10h
3E76	LD A,118
D7	RST 10h
18F8	JR START

Her får du en lodret række af grafiktegnene. Det skyldes, at når A-registeret indeholder 118 (Newline), vil en RST 10h simpelt hen skifte linie. I det næste eksempel bruger vi både RST 8h og RST 10h:

Heks-kode	Mnemonic
FD362200	LD (IY+34),0
010003	LD BC,768
LØKKE	
3A3440	LD A,(16436)
CB77	BIT 6,A
20F9	JR NZ, LØKKE
D7	RST 10h
0B	DEC BC
78	LD A,B
B1	OR C
20F3	JR NZ, LØKKE
CF	RST 8h
04	DEFB

I de følgende linier forklarer vi programmet. Hvis du ikke har problemer med forståelsen, kan du blot springe forklaringen over!

Vi vil lige fortælle, at systemvariabel 16418 indeholder det antal linier, som ZX 81 bruger til rapport, programmering, editering etc. Denne er sædvanligvis 2 (de to nederste linier), medmindre det bliver ændret. Det er netop, hvad vi gør med

```
LD (IY+34),0
```

Dette gøres, så vi kan bruge hele skærmen, inkl. de to nederste linier. Derefter LOAD'er vi BC med antallet af pladser på skærmen, og det er 24 linier á 32 kolonner, i alt 768 (eksklusive N/L, RST 10h går jo uden om disse). Derefter står der LD A,(16436). Systemvariablen FRAMES på adresse 16436 tæller antallet af skærmbilleder. Den bliver formindsket med 50 i sekundet, da der netop vises 50 billeder pr. sekund. Vi får altså et nogenlunde »tilfældigt« tal. Derefter testes med BIT 6,A, om tegnet er displaybart.

Hvis ikke må vi have et nyt tal i A, da ZX 81 ellers vil crashe ved udførelse af en RST 10h-instruktion. Når et displaybar tegn er fundet, bliver det skrevet med RST 10h, svarende til CHR$ A. Derefter undersøges, om skærmen er fyldt med

```
DEC BC
LD A,B
OR C
```

Hvis BC er 0, er skærmen fyldt, hvis ikke tager den en tur til. Hvis den er fyldt, fortsættes med RST 8h, som vi rimeligvis får til at udskrive kode 5 (SF), der er fyldt skærm. Det er selvfølgelig ikke nødvendigt at undersøge, om skærmen er fyldt, men det hjælper forhåbentlig på forståelsen af RST 8h-instruktionen.

RST 18h og RST 20h vil vi ikke forklare, da begynderen ganske enkelt ikke kan bruge disse to instruktioner. Dog kan vi fortælle, at de begge bruges flittigt i ROM's arbejde, når vi arbejder i BASIC, men vi kan ikke drage nytte af dem.

RST 28h er en rutine, som vi kan benytte under matematiske beregninger. Denne instruktion er så omfattende, at vi har henlagt den til et senere kapitel.

RST 30h er en rutine, der sørger for, at der skabes plads til BASIC-linier. Vi vil heller ikke forklare denne instruktion.

RST 38h er endnu en uforklarlig instruktion. Vi kalder den »display-interrupt«-rutinen, og denne vil vi heller ikke beskæftige os med.

Kapitel 13

I dette kapitel vil vi gennemgå nogle instruktioner, som opererer med hele blokke af data. Den første er LDI, LoaD Increment, der virker som følger:

```
LD (DE),(HL)
INC DE
INC HL
DEC BC
```

Hele dette lille »program« er altså gemt i en instruktion. Vi vil gerne gøre opmærksom på, at LD (DE),(HL) ikke findes. Eksempel:

Heks-kode	**Mnemonic**
112F75	LD DE,29999
210000	LD HL,0
010100	LD BC,1
EDA0	LDI
C9	RET

Skriv derefter PRINT USR 30000. DE er nu blevet 30000, og HL er 1. BC bliver skrevet med PRINT USR, og svaret er 0, idet BC jo var 1, før den blev decrementeret af LDI. Prøv nu at skrive PRINT PEEK 29999. Resultatet vil være 211. Hvorfor? Fordi der ville blive POKE'et på DE med PEEK HL, og PEEK HL var jo netop 211, fordi HL var 0. P/V-flaget blev i øvrigt resat, da BC blev 0.

Hvad nu, hvis man ville have overført f.eks. 0 til 99 til 29000 til 29099. Så måtte man gøre følgende :

```
LD DE,29000
LD HL,0
LDI
LDI
LDI ... i alt 100 gange
```

Dette er tydeligvis en meget besværlig operation. Derfor har man en lignende instruktion, LDIR (LoaD Increment Repeat), blot med den udvidelse, at denne repeterer, indtil BC er nul. Nu kan vores program laves således:

Heks-kode	Mnemonic
114871	LD DE,29000
210000	LD HL,0
016400	LD BC,100
EDB0	LDIR
C9	RET

Efter RAND USR 30000 kan du som kontrol prøve, om PEEK 29000 er lig PEEK 0, PEEK 29001 lig PEEK 1, osv.

Prøv dette lidt mere praktisk. Lav en linie 1 i BASIC med indholdet REM Z80A-PROCESSOR. Det er vigtigt, at det er den allerførste linie i programmet. Når den er det, vil Z i Z80A stå på plads 16514 (17307). Derefter indtastes:

Heks-kode	Mnemonic	Forklaring
ED5B0C40	LD DE,(16396)	Identificer udskriftfilen
(117D40 – LD DE,16509)		
13	INC DE	Første skærmposition
218240	LD HL,16514	Starten af Z80A...
(219B43 – LD HL,17307)		
010E00	LD BC,14	Der er 14 tegn
EDB0	LDIR	Kopiér til skærmen
C9	RET	Returner til BASIC

Skriv RAND USR 30000.

I dette program overføres tegnet på adresse 16514 (17307), dvs. Z, til første punkt på skærmen, og der rykkes frem til andet tegn på andet punkt, osv., i alt 14 gange. Der kan laves flere gode programmer med denne instruktion. Man kan f.eks. gemme et skærmbillede. Dette gøres med følgende program:

Heks-kode	Mnemonic	Adresse
119475	LD DE,30100	30000
2A0C40	LD HL,(16396)	30003
(217D40 – LD HL,16509)		
011903	LD BC,793	30006
EDB0	LDIR	30009
C9	RET	30011
ED5B0C40	LD DE,(16396)	30012
(117D40 – LD DE,16509)		
219475	LD HL,30100	30016 (30015)
011903	LD BC,793	30019 (30018)
EDB0	LDIR	30022 (30021)
C9	RET	30024 (30023)

Derefter laver du følgende BASIC-program:

```
100 FOR N=1 TO 50
110 PRINT AT RND*21, RND*31; CHR$ (RND*63)
120 NEXT N
130 RAND USR 30000
```

Efter at programmet er færdigt, er hele skærmbilledet blevet gemt på adresserne fra 30100 til 30892. Når du ønsker billedet fremkaldt, skriver du RAND USR 30012.

Vi håber, at du selv kan forstå programmet. Ellers må du repetere lidt.

Et andet program, man kan lave med LDIR-ordren, er at scrolle skærmen. Da SCROLL-rutinen i ROM giver visse problemer, når skærmen skal slettes, vil vi selv give opskriften på en SCROLL-rutine.

Heks-kode	Mnemonic
ED5B0C40	LD DE,(16396)
(117D40 – LD DE,16509)	
212100	LD HL,33
19	ADD HL,DE
01D602	LD BC,726
EDB0	LDIR
C9	RET

Du vil med denne rutine få scrollet skærmbilledet opad, og du kan naturligvis kalde den fra BASIC.

Først LOAD'es DE med udskriftfilens begyndelsesadresse, hvorefter vi LOAD'er HL med 33, og lægger DE's indhold til HL.

Nu peger HL på 1. punkt på 2. linie. LDI-instruktionen udføres 726 gange, svarende til 22 linier á 33 tegn. Denne gang lader vi de to nederste linier ligge uberørt. LDIR vil først flytte 2. linie's 1. tegn, i (HL), til 1. linie's 1. tegn, i (DE), osv. Hele skærmbilledet vil altså blive rykket en linie op. Bemærk, at vi faktisk også POKE'er på N/L's adresse, men det vi POKE'er, er jo en anden N/L.

Vi har nu gennemgået to af blokinstruktionerne. Faktisk er der to mere i samme kategori; de har kun en lille, men væsentlig forskel.

LDD betyder LoaD Decrement, og den udfører dette lille program:

```
LD (DE),(HL)
DEC DE
DEC HL
DEC BC
```

Som du kan se, vil DE og HL blive decrementeret i modsætning til LDI, hvor de incrementeres.

Du har vel allerede nu gættet, at der findes en 4. instruktion, der hedder LDDR - LoaD Decrement Repeat – som repeterer, til BC er talt ned til 0.

Ved alle disse instruktioner vil P/V-flaget blive resat, hvis BC-

registeret tælles ned til 0. LDIR og LDDR vil derfor altid medføre, at P/V-flaget bliver resat.

Vi vil ikke komme med nogle små eksempler. Vi håber, at du har forstået instruktionen, da vi nu vil bruge den i endnu en SCROLL-rutine. Denne rutine skal rulle billedet nedad!!!

Vi opfordrer dig til selv at lave denne rutine, da du nu bør kunne klare en sådan opgave. Et tip skal du dog have med på vejen: brug LDDR. Hvis du laver en fejl eller to, behøver du ikke fortvivle; øvelse gør mester, som bekendt.

Hvis du kører fast i opgaven, så læs trygt videre. Nu vil vi gennemgå problematikken i denne opgave.

Først må vi gøre os det klart, at tegnet på den adresse, som HL peger på, skal kopieres på adressen, DE peger på. Da skærmbilledet skal rulle nedad, må DE antage den største værdi. Vi undtager atter de to nederste linier – det kan du selv ændre, hvis du vil. 22. linie, den sidste, har N/L liggende på position 792 minus 66, hvilket er 726, efter udskriftfilens start. På 725. position skal det første »scrollede« tegn stå. Det scrolles fra position 725 minus 33, der er 692, og det må være HL's position.

Vores første skridt er nu at identificere udskriftfilen og addere 692 for at finde frem til HL's position.

```
LD HL,(16396)
LD BC,692
PUSH BC
ADD HL,BC
```

Denne position er nu fastlagt. Nu skal vi have fundet DE's position, og den er præcis 33 mere. Da 16 bits-addition kun kan foregå til HL-registeret, må DE's værdi først findes i HL, hvorefter den overføres til DE-registeret. Vi ønsker jo imidlertid at beholde HL's indhold, så vi lægger det op på stakken.

```
PUSH HL
```

Nu kan vi lægge 33 til.

```
LD BC,33
ADD HL,BC
```

Dette resultat skal nu overføres til DE. Da LD DE,HL ikke findes, er den nemmeste overførsel EX DE,HL. Her bytter DE og HL jo værdier. Hvad kommer nu i HL? Det ved vi ikke, da vi ikke kender DE's indhold, men hovedsagen er, at det er betydningsløst! Vi vil jo alligevel straks hente den gamle HL-værdi tilbage fra stakken.

```
EX DE,HL
POP HL
```

Så mangler vi kun at finde en værdi til BC. Dette register indeholder altid antallet af gennemløb, og det må i dette tilfælde være 692, nemlig 21 linier á 33 tegn-Newline. Den nederste linie er jo væk, scrollet ud af skærmen. Imidlertid har vi i starten af programmet LOAD'et BC med 692, så hvorfor LOAD'e to gange? I stedet lægger vi værdien oven på stakken i starten af programmet, så vi senere kan hente den ned.

```
tidl: PUSH BC
```

Nu henter vi den ned med

```
POP BC
```

Alt er parat. Nu kan vi scrolle med

```
LDDR
```

Husk, at vi starter i nederste højre hjørne, og derfor slutter i øverste venstre.

Er vi så færdige? Nej, ikke helt, for den øverste linie på skærmen, der er scrollet ned til 2. linie, skal slettes. Til dette har vi atter brug for at kunne identificere udskriftfilen. Faktisk kan vi gøre det utroligt hurtigt. HL-registeret pegede i sidste gennemløb på 1. tegn på skærmen. Så blev det decrementeret, hvorefter

LDDR var færdig. Nu peger det på den 1. N/L i udskriftfilen. Vi skal løbe en løkke igennem, der fjerner et tegn 32 gange. Det kan gøres således:

```
LD B,32
INC HL
LD (HL),0
DJNZ –5
```

Bemærk, at vi først incrementerer HL – for at undgå N/L – før vi fjerner tegnet. Ved DJNZ skal vi tilbage til INC HL, altså –5.

Vores program ser – med en RET-instruktion tilføjet – således

Heks-kode		Mnemonic	Adresse
2A0C40		LD HL,(16396)	30000
(217D40 – LD HL,16509)			
01B402		LD BC,692	30003
C5		PUSH BC	30006
09		ADD HL,BC	30007
E5		PUSH HL	30008
012100		LD BC,33	30009
09		ADD HL,BC	30012
EB		EX DE,HL	30013
E1		POP HL	30014
C1		POP BC	30015
EDB8		LDDR	30016
0620		LD B,32	30018
23	OM	INC HL	30020
3600		LD (HL),0	30021
10FB		DJNZ OM	30023
C9		RET	30025

Kapitel 14

Der findes flere blok-instruktioner. De fire, som vi nu vil gennemgå, er i familie med CP.

Den første er CPI (ComPare Increment), som udfører følgende lille program:

```
CP (HL)
INC HL
DEC BC
```

Indholdet i register A og i (HL) sammenlignes. HL incrementeres, og BC decrementeres. Hvis A og (HL) var ens, sættes Z-flaget! S-flaget virker på sædvanlig vis. C-flaget derimod berøres *ikke* af CPI. Når vi ikke kan bruge C-flaget, kan vi heller ikke finde ud af, om (HL) er større eller mindre end A's indhold. P/V-flaget virker på samme måde som i de forrige blok-instruktioner. Hvis BC er blevet nul, resættes P/V-flaget. Der hører også en instruktion med repetition til samlingen, nemlig CPIR (ComPare Increment Repeat), og den virker som CPI, men repeterer til BC er nul. Når CPIR er færdig med sit arbejde, kan den ene grund være, at den søgte værdi er fundet. Hvis den er det, vil Z-flaget være sat, og værdien står på byte HL–1. Husk at HL incrementeres efter CP (HL). Hvis Z-flaget er nul efter CPIR, er værdien ikke fundet. Den værdi, der søges efter, er naturligvis lagt i A-registeret, da det jo er dette register, der sammenlignes med (HL).

Vi vil lige illustrere CPIR-instruktionen i et par eksempler. I det første eksempel vil vi vende hvert mellemrum på skærmen. Man behøver ikke benytte CPIR-instrutionen til denne simple opgave. Det kan gøres nemmere, men ikke desto mindre er det et godt eksempel på ordrens virkemåde.

Heks-kode	Mnemonic	Forklaring
2A0C40	LD HL,(16396)	Udskriftfilen findes
(217D40 – LD HL, 16509)		
011803	LD BC,792	Udskriftfilens størrelse i BC
3E00	LD A,0	Den søgte værdi (mellemrum)
23 GENTAG	INC HL	Peg på næste tegn
EDB1	CPIR	Undersøg for mellemrum
E0	RET PO	Returner, hvis hele skærmen er gennemsøgt
2B	DEC HL	Gå tilbage til den fundne værdi
CBFE	SET 7,(HL)	Vend mellemrummet
18F7	JR GENTAG	Hop til næste tegn

Læg mærke til, at vi bruger RET PO. Der returneres kun, hvis BC er talt ned til nul, eftersom P/V-flaget jo resættes, når BC er nul. PO står som bekendt for Parity Odd, og det er P/V-flaget lig nul. Vi skriver i programmet INC HL, fordi vi før omvendingen har lavet en DEC HL, og der skal jo fortsættes fra det sted på skærmen, hvor vi sluttede. DEC HL er med, fordi det er byten på HL-1, der skal have sat bit 7. Denne bit gør jo karakteren hvid på sort baggrund. Bemærk, at vi starter på den 1. position på skærmen ved sammenligningen.

Vort 2. eksempel:

Heks-kode	Mnemonic	Forklaring
210000	LD HL,0	Start på adresse 0
010020	LD BC,8192	Gennemsøg maksimum 8192 bytes
3E0D	LD A,13	Søg efter værdien 13
EDB1	CPIR	Sammenlign
C0	RET NZ	Returner, hvis 13 ikke findes
2B	DEC HL	Gå tilbage til byten med 13
44	LD B,H	Overfør BC
4D	LD C,L	til HL
C9	RET	Returner til BASIC

PRINT USR 30000. Derefter skulle udskriften blive 56. Prøv efter med PRINT PEEK 56 i BASIC. Blev svaret 13? Adresse 56 er altså første gang, værdien 13 findes i ROM.

Naturligvis findes også CPD-instruktionen. Her decrementeres HL i stedet for at blive incrementeret.

```
CP (HL)
DEC HL
DEC BC
```

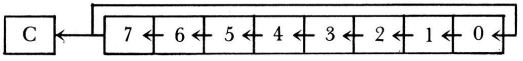
Og – som du vel har gættet – så findes CPDR, der repeterer til BC er nul. Proceduren er den samme, og flagsætningen fungerer også på samme måde som de andre CP-blok-instruktioner.

Vi vil overlade det til dig selv at eksperimentere videre med CPD og CPDR. Jo mere du øver dig i at lave programmer i maskinkode, jo bedre bliver du.

Kapitel 15

I Z80-sproget findes nogle såkaldte rotations- og skifte-instruktioner. Disse instruktioner roterer rundt med bittene i et af følgende registre: A, B, C, D, E, H, L, (HL), (IX+d) eller (IY+d). Hvordan de virker vises bedst skematisk.

Den første er RLC-instruktionen (Rotate Left Carry), og den virker således:



Bit 7 til bit 0
Bit 7 til Carry

Her er C carry-flaget og ikke register C. Hvis A-registeret indeholder 10011010b, og C-flaget er nul, vil det efter en RLC se således ud: 00110101 og carry-flaget 1. Bit 7, som er sat, er rykket over til carry-flaget, der således er blevet sat. Bit 0 er rykket til bit 1, som er rykket til bit 2, osv., og bit 7 er rykket om til bit 0. For at illustrere det på skærmen, vil vi lave et program:

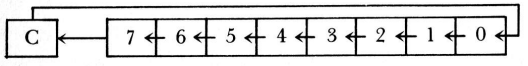
Heks-kode	Mnemonic	Forklaring
2A0C40	LD HL,(16396)	Identificer udskriftfilen
(217D40 – LD HL,16509)		
23	INC HL	Peg på 1. position på skærmen
3612	LD (HL),18	Skriv karakteren med kode 18 dér
113075	LD DE,30000	Pause
1B	DEC DE	
7B	LD A,E	
B2	OR D	
20FB	JR NZ,–5	

CB06	RLC (HL)	Roter koden 18
C9	RET	Returner til BASIC

Efter at have kørt programmet vil du se, at der i hjørnet blev skrevet et 8-tal oven i et »større end«. Hvorfor? I starten POKE'r vi 18 – det er »større end« – ind på skærmen. Derfor holdes den sædvanlige pause, så du kan se, hvad der sker. Derefter kommer RLC (HL). 18 er binært 00010010. Derefter roteres til venstre, så der nu vil stå 00100100. Dette er 36 i decimal, og ved nærmere undersøgelser finder man ud af, at koden for 8-tallet netop er 36. Hvad C-flaget indeholder er ligegyldigt, da vi ikke skal bruge det. Prøv selv at eksperimentere videre med dette.

Der er flere rotations-instruktioner. De vil også blive vist skematisk, men der vil ikke komme programeksempler til dem. De virker på næsten samme måde, så vi opfordrer dig til selv at afprøve dem.

RL (Rotate Left):

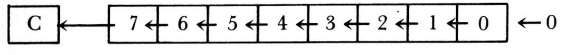


Bit 7 til Carry
Carry til bit 0

Læg mærke til, at det her ikke er bit 7, der går til bit 0, men carry-flaget, der vandrer derover.
Eksempel:

Register A:	01010100	C=1
A efter RL A:	10101001	C=0

SLA (Shift Left Arithmetic):

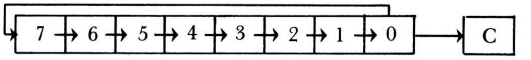


Bit 7 til Carry
Bit 0 resættes

Eksempel:

Register A:	10101001	C=0
A efter SLA A:	01010010	C=1

RRC (Rotate Right Carry):

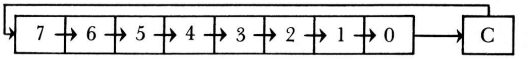


Bit 0 til Carry
Bit 0 til bit 7

Læg mærke til, at der roteres baglæns med RRC.
Eksempel:

Register B:	10101010	C=1
B efter RRC B:	01010101	C=0

RR (Rotate Right):

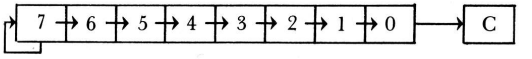


Bit 0 til Carry
Carry til bit 7

Læg mærke til, at RRC ikke er det samme som RR C.
Eksempel:

Register C:	10101010	C=1
C efter RR C:	11010101	C=0

SRA (Shift Right Arithmetic):

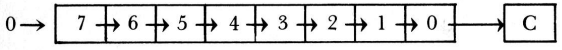


Bit 0 til Carry
Bit 7 berøres ikke

Eksempel:

Register D:	10010011	C=0
D efter SRA D:	11001001	C=1

SRL (Shift Right Logic):



Bit 0 til Carry
Bit 7 resættes

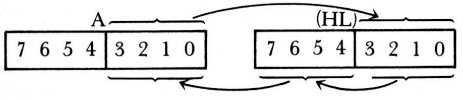
Eksempel:

Register D:	10010011	C=1
D efter SRL D:	01001001	C=1

Bid mærke i, at SRA og SRL laver det samme, bortset fra at SRL nulstiller bit 7.

De to sidste rotationsinstruktioner er lidt anderledes. De virker kun på register A og (HL) – altså den byte, HL peger på.

RLD (Rotate Left Digit):



Hvis du ikke har gennemskuet tegningen helt, må du lige studere denne forklaring. Bit 0, 1, 2, og 3 i A-registeret overføres til bit 0, 1, 2 og 3 i (HL), altså bit 0 i A går til bit 0 i (HL), bit 1 i A til bit 1 i (HL), osv. Bit 0, 1, 2 og 3 i (HL) går til bit 4, 5, 6 og 7 også i (HL). Endelig går bit 4, 5, 6, 7 i (HL) til bit 0, 1, 2 og 3 i A. Dette må vi hellere vise med et eksempel:

Heks-kode	**Mnemonic**
3E96	LD A,150
212F75	LD HL,29999

77	LD (HL),A
3EFF	LD A,255
ED6F	RLD
0600	LD B,0
4F	LD C,A
C9	RET

Prøv at køre dette program med PRINT USR 30000, PEEK 29999. Det første tal, 249, er det A indeholder efter RLD! Det sidste er, hvad (HL) indeholder efter. Vi starter forfra i binær notation.

A indeholdt 255, og (HL) indeholdt 150

A: 1111 1111 (HL): 1001 0110

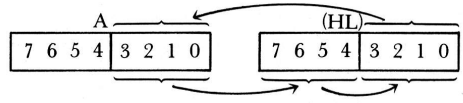
Efter at RLD har gjort sit arbejde med sin forunderlige ombytning, ser A og (HL) således ud:

A: 1111 1001 (HL): 0110 1111

Find selv ud af, at det er forløbet korrekt. A er i decimal 249, og (HL) er 111. Netop disse tal fik du skrevet ud på skærmen. Nu ved vi, at RLD virker som fortalt.

Der findes endnu en instruktion af samme slags, nemlig RRD, som også virker på A og (HL), men på følgende måde:

RRD (Rotate Right Digit):



Vi afprøver med dette program, men du må ikke køre det endnu:

Heks-kode	**Mnemonic**
3E96	LD A,150
212F75	LD HL,29999
77	LD (HL),A

```
3EFF        LD A,255
ED67        RRD
0600        LD B,0
4F          LD C,A
C9          RET
```

Inden det køres med PRINT USR 30000, PEEK 29999, vil vi selv finde ud af, hvad der bliver udskrevet.

A indeholdt 255 og (HL) 150

A: 1111 1111 (HL): 1001 0110

Efter RRD ser det således ud:

A: 1111 0110 (HL): 1111 1001

A-registerets indhold er i decimal 246, (HL) er 249. Selve rokeringen af de forskellige bits bør fremgå af vor illustration af RRD-instruktionen.

Efter PRINT USR 30000, PEEK 29999 skal udskriften altså være 246 og 249, og det bliver den vel også?

I skemaet herunder står alle heks-koderne for rotationsordrerne. Husk: Foran alle skal der være CB, undtaget IX og IY-rotationsinstruktioner, hvor CB bliver 2. byte i koden. Rotationsordrerne fylder to bytes (IX og IY dog fire bytes).

	B	C	D	E	H	L	(HL)	A	(IX+d)	(IY+d)
RLC	00	01	02	03	04	05	06	07	DD06	FD06
RRC	08	09	0A	0B	0C	0D	0E	0F	DD0E	FD0E
RL	10	11	12	13	14	15	16	17	DD16	FD16
RR	18	19	1A	1B	1C	1D	1E	1F	DD1E	FD1E
SLA	20	21	22	23	24	25	26	27	DD26	FD26
SRA	28	29	2A	2B	2C	2D	2E	2F	DD2E	FD2E
SRL	38	39	3A	3B	3C	3D	3E	3F	DD3E	FD3F

For eksempel er SLA L heks-kode CB25, RR (HL) hedder CB1E, og RRC (IX+55) er DDCB370E (55 er 37h). Læg mærke til, at d er klemt inde som 3. tal i den faste kode. Det er det altid! SRA (IY–100 er FDCB9C2E i heks-kode. Igen er –100 klemt inde, og du husker vel, at negative tal udtrykkes ved 256 – tallet positivt. I vores tilfælde 256 – 100, og det er 156, 9Ch.

I tabellen står ikke de to »store« rotationsinstruktioner, men de er her:

Heks-kode	Mnemonic
ED67	RRD
ED6F	RLD

Inden vi laver et program til at vise brugen af rotation, skal det lige bemærkes, at der findes 4 instruktioner. RRCA svarende til RRC A, RLCA lig RLC A, RLA lig RL A, og endelig RRA svarende til RR A. Deres heks-koder er her:

Heks-kode	Mnemonic
07	RLCA
0F	RRCA
17	RLA
1F	RRA

Der skal ikke CBh foran, men de virker præcis som med CB foran. Alligevel har Zilog medtaget disse 4 en-byte-instruktioner opererende på A-registeret.

Vi vil lave et program til ZX 81 (ikke Lambda), som udskriver alle displaybare tegn i forstørret tilstand, faktisk 64 gange. Du skal lige vide, at tegngeneratoren, det sted i ROM, hvor tegnenes udseende står, starter på 1E00h, der er adresse 7680. Et tegns udseende fylder 8 bytes. Hvis du vil vide, hvordan et 2-tal ser ud, finder du først ud af, hvor 2-tallet ligger. Det gøres med 7680 + 8*CODE »2«, og det er 7920. Hvis du PEEK'er de 8 bytes fra 7920 til og med 7927, finder du tallene 0, 60, 66, 2, 60, 64, 126 og 0. I binær notation ser det således ud:

```
7920    indhold: 0      binært: 00000000
7921    indhold: 60     binært: 00111100
7922    indhold: 66     binært: 01000010
7923    indhold: 2      binært: 00000010
7924    indhold: 60     binært: 00111100
7925    indhold: 64     binært: 01000000
7926    indhold: 126    binært: 01111110
7927    indhold: 0      binært: 00000000
```

Hvis du fjerner nullerne og kun betragter ettallerne, ser mønsteret således ud:

```
  1111
1    1
     1
 1111
1
111111
```

Og det ligner jo ganske rigtigt et 2-tal.

Det vi vil gøre i programmet er simpelt hen at trække en byte ind ad gangen fra tegngeneratoren. Derefter checker vi, om bit 7 er sat. Hvis ja, så udskriv en sort firkant – hvis nej, udskriv en hvid firkant. Så testes bit 6, osv., helt ned til bit 0. Så scroller vi skærmbilledet, hvorefter vi tager næste byte i generatoren. Når der ikke er flere tegn returnerer vi dér. Selve tegngeneratoren slutter på adresse 8191.

Først vil vi lave den del, som undersøger bittene. Vi LOAD'er A med (HL), som peger på den byte, vi er nået til (HL defineres senere):

```
LD A (HL)
```

Der er 8 bits i en byte, og de skal alle testes. Det gøres nemmest med en DJNZ-løkke.

```
LD B,8
```


DJNZ følger senere. En smart måde at undersøge bittene på er at køre dem én efter én ud til carry, og det gøres med RLC A, der jo findes som RLCA en byte kortere.

```
LØKKE    RLCA
```

Først gemmes A; så undersøges det, om carry-flaget er blevet sat. Hvis det er sat, udskrives sort firkant, ellers en hvid. Sort har koden 128. Husk at A slettes (nulstilles) med XOR A.

```
PUSH AF
LD A,128
JR C,1
XOR A
RST 10h
POP AF
```

Er der flere utestede bits i A, så hoppes til RLCA fra

```
DJNZ LØKKE
```

Og derefter returneres til hovedrutinen.

```
RET
```

Dette var en underrutine. Vi har også brug for en rutine til at scrolle, men vi udvider den en smule. For det første ønsker vi at gemme HL, som holder styr på, hvilken byte vi er nået til.

```
PUSH HL
```

Derefter følger det gamle SCROLL-program:

```
LD DE,(16396)
PUSH DE
LD HL,33
ADD HL,DE
LD BC,726
LDIR
```

Læg mærke til, at der lige er puttet en PUSH DE ind foroven. DE indeholder jo udskriftfilens start, og den skal vi bruge nu, så

```
POP HL
```

Nu indeholder HL udskriftfilens start, og til den lægger vi 694. Men hvorfor? Fordi RST 10h jo skriver på den næste printposition, og vi ønsker, at der skal skrives dér efter en scroll-ordre. Det, der står der i forvejen, er jo scrollet i sikkerhed. Udskriftfilens begyndelsesadresse + 694 svarer til den position i BASIC, som vi kalder AT 21,0. I systemvariablen 16398 + 256 * PEEK 16399 står adressen på den næste printposition, så derfor:

```
LD BC,694
ADD HL,BC
LD (16398),HL
```

Vi kan måske også få brug for et STILL-billede, så vi vil gøre det på den måde, at hvis BREAK er aktiveret, stopper den, indtil man slipper tasten. Faktisk køres en tom løkke:

```
CALL 3910
JR NC,-5
```

Derefter POP'er vi HL tilbage, og HL er jo den, der holder styr på, hvilken byte vi er nået til. Du husker, at vi PUSH'ede HL i starten. Den skal vi bruge igen:

```
POP HL
RET
```

Derefter returnerer vi til hovedrutinen. Den skal vi lave nu. Det første er, at HL skal LOAD'es med starten af tegngeneratoren.

```
LD HL,7680
```

Derefter kalder vi vores SCROLL-rutine.


```
CALL SCROLL
```

Så kalder vi BIT-testen.

```
CALL BIT-TEST
```

Næste byte findes med

```
INC HL
```

Hvis HL er blevet incrementeret til 8192, er der ingen tegn, og der skal returneres. Hvis HL er nået til 8192, er bit 5 i H sat – og kun da!

```
BIT 5,H
RET NZ
JR -12
```

Hvis den er sat, vil Z-flaget resættes, da dette flag bliver sat til den modsatte værdi af den testede bit. Herved udføres RET NZ. Til sidst skal der køres om, hvis der er flere.

Programmet, der ikke virker på Lambda, er i sin helhed:

Heks-kode		Mnemonic	Adresse
7E	BIT-TEST	LD A,(HL)	30000
0608		LD B,8	30001
07	LØKKE	RLCA	30003
F5		PUSH AF	30004
3E80		LD A,128	30005
3801		JR C,1	30007
AF		XOR A	30009
D7		RST 10h	30010
F1		POP AF	30011
10F5		DJNZ LØKKE	30012
C9		RET	30014
E5	SCROLL	PUSH HL	30015
ED5B0C40		LD DE,(16396)	30016

```
D5                      PUSH DE         30020
212100                  LD HL,33        30021
19                      ADD HL,DE       30024
01D602                  LD BC,726       30025
EDB0                    LDIR            30028
E1                      POP HL          30030
01B602                  LD BC,6,9       30031
09                      ADD HL,BC       30034
220E40                  LD (16398),HL   30035
CD460F      BREAK       CALL 3910       30038
30FB                    JR NC,BREAK     30041
E1                      POP HL          30043
C9                      RET             30044
-------------------------------------------------
21001E      START       LD HL,7680      30045
CD3F75      FORFRA      CALL SCROLL     30048
CD3075                  CALL BIT-TEST   30051
23                      INC HL          20054
CB6C                    BIT 5,H         30055
C0                      RET NZ          30057
18F4                    JR FORFRA       30058
```

Programmet er lagt fra adresse 30000. Hvis du ændrer dette, skal alle CALL-adresser naturligvis også ændres. Programmet startes med RAND USR 30045, hvorefter START-løkken begynder. Du kan holde billedet fast med BREAK. Hvis du ønsker omvendte tegn, laver du JR C,1 i første rutine om til JR NC, 1. Dette gøres ved POKE 30007, 48, idet 48 er 30h, og det er hekskoden for JR NC. Prøv også at bytte RLCA om med RRCA. Det gøres med POKE 30003,15. Hvorfor gør programmet sådan nu?

Prøv også selv at ændre programmet, så udskriften bliver mere elegant. F.eks. kan du lade de forskellige tegn være adskilt fra hinanden i kasser lavet af stjerner, og du kan indlægge pauser i stedet for fastfrysningen med BREAK. Der er muligheder nok.

Kapitel 16

Der findes to EXchange-instruktioner, som vi endnu ikke har omtalt. Det drejer sig om

Heks-kode	Mnemonic
08	EX AF,AF'
D9	EXX

Inden vi forklarer selve instruktionerne, så vil vi fortælle dig om opbygningen af Z80-registersættet.

Alle registrene, A, B, C, D, E, H og L, har et »tvilling«-register. Hvert af registrene består derfor af to registersæt, hvoraf det ene »sover«, mens det andet udfører dagens dont. Med ordren EXX kan man bytte om på de to register-sæt, så det fungerende får sig en hvilepause, mens det andet bliver vækket. Det »passive« register til BC kaldes BC', DE's kaldes DE', HL's kaldes HL', og endelig kaldes AF's »passive« register AF'. Ordren EXX udfører tre ting:

EX BC,BC'
EX DE,DE'
EX HL,HL' } findes ikke som selvstændige instruktioner

Den anden instruktion, EX AF,AF', ombytter naturligvis blot AF med AF'.

Når et register gøres passivt, tager det sin værdi »med i seng«, og hvis man henter det tilbage, kommer den »gamle« værdi fra lagringstiden med op til nutidens vilde liv.

Ønsker man at gemme BC, DE, HL – f.eks. på stakken – for derefter at tage dem frem samlet, så vil EXX være nemmere. Se her:

Metode 1	Metode 2
PUSH BC	
PUSH DE	EXX
PUSH HL	
...	...
opgave	opgave
...	...
POP HL	
POP DE	EXX
POP BC	

Programmet til højre vil være kortere.

Man skal dog være lidt forsigtig med EX AF,AF'. Den fungerer kun i FAST-tilstand, da ZX 81 ikke er glad for, at man bytter rundt på registrene. Skærmbilledrutinen i ROM bruger AF', og derfor tillades kun FAST.

Eksempel på brugen af instruktionerne:

Heks-kode	Mnemonic	Forklaring
D9	EXX	Registrene ombyttes
01E803	LD BC,1000	Læg værdien 1000 i BC
D9	EXX	Ombyt registrene igen
C9	RET	Returner til BASIC

Vi skriver PRINT USR 30000. Svaret er 30000 – både i FAST og SLOW. Hvorfor er den det? Vi husker fra tidligere, at med USR-funktionen LOAD'es BC med tallet efter, her 30000. Herefter gemmer vi BC, DE og HL og kalder deres respektive passive tvillinger frem. Nu LOAD'er vi BC med 1000. Herefter bytter vi igen rundt. Det oprindelige BC-register kan komme tilbage (indeholdende sin oprindelige værdi, der er 30000), og dets passive modstykke lægges tilbage, hvor det kom fra. Der returneres, og BC-værdien udskrives. Den er naturligvis 30000.

Dette kan bidrage til effektiviteten i dine programmer. Det må dog kraftigt anbefales at lade være med at bruge EXX, hvis du bruger ROM-rutiner eller RST.

Som vi før har omtalt, bruges N- og H-flagene til instruktionen DAA. Denne er til brug for BCD-aritmetik (Binary Coded Decimal). I 4 bits kan 16 forskellige værdier udtrykkes (2 gange 2 gange 2 gange 2), og vi kalder dem fra 0 til F. I BCD-systemet ser man bort fra A til F, og arbejder kun med cifrene 0 til 9. En liste over cifrene ser således ud:

Heksa-decimalt	Binært	
0	0000	
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
A	1010	Bruges ikke i BCD-system
B	1011	
C	1100	
D	1101	
E	1110	
F	1111	

Det største BCD-ciffer er 1001b, og derfor vil det højeste BCD-tal (8 bits) være 10011001b, eller 99 i decimal notation.

Vi tager et eksempel:

```
  21 h
+ 36 h
  57 h
```

Her vil 21 + 36 være 57, og betragter vi 21 og 36 som to decimale tal passer regnskabet også (33 + 54 = 87).

Men:

```
  32 h
+ 59 h
  8B h
```

Og betragtet decimalt giver 32 + 59 lig 8B ingen mening. Det er 91! Dette resultat får vi frem ved – efter udregningen – at bruge DAA-instruktionen.

Man kan også trække fra med BCD-aritmetik. Kun instruktionerne ADD, ADC, SUB, SBC, INC, DEC og NEG virker, når man arbejder med BCD-systemet.

H- og N-flaget benytter Z80A til at konvertere rigtigt ved DAA-instruktionen. H-flaget, halfcarry-flaget, sættes, hvis en 8 bits-addition medfører mente til, eller subtraktion medfører lån fra bit 4 i register A. Hvor C-flaget indikerer mente til /lån fra bit 8 (9. bit), indikerer H-flaget mente til/lån fra 5. bit. N-flaget sættes, hvis den foregående instruktion var en subtraktion. Du kan ikke selv bruge H- og N-flaget i dine programmer, da du ikke kan teste dem til brug i betingede instruktioner. Flagene sættes hhv. resættes naturligvis også ved andre instruktioner.

DAA-instruktionen påvirker i øvrigt flagene på helt naturlig vis, og P/V-flaget virker her som paritetsindikator.

BCD-aritmetik bruges mest i forbindelse med økonomi- og finansprogrammer, da det dér er velegnet at bruge 99 som det største tal. Systemet benyttes ofte i forbindelse med de to rotationsinstruktioner RLD og RRD, men systemet vil ikke blive gennemgået her. Vi vil gerne understrege, at det er helt op til dig selv, om du vil arbejde i BCD-systemet, og du skal formentlig have lidt mere overblik, før du kan benytte det.

*

Har du tallet 01110101b og ønsker det omsat til heksadecimal notation, deler du byten på midten. 0111b er 7h, og 0101b er 5h. Den samlede heks-kode er 75h.

Vi tager et eksempel. Hvis 11011001b ønskes omsat, deler vi og får hjælp af tabellen i appendiks C

1101 = D
1001 = 9

og vi finder, at 11011001b er D9h.

Som et lille sidespring kan vi med denne viden lave en »om-

vendt« heks-loader, der udskriver heksadecimale tal fra bytes i ROM eller RAM efter ønske.

Heks-kode	Mnemonic
213075	LD HL,30000
061E	LD B,30
7E	LD A,(HL)
F5	PUSH AF
E6F0	AND 240
1F	RRA
1F	RRA
1F	RRA
1F	RRA
C61C	ADD 28
D7	RST 10h
F1	POP AF
E60F	AND 15
C61C	ADD 28
D7	RST 10h
3E00	LD A,0
D7	RST 10h
D7	RST 10h
23	INC HL
10E8	DJNZ –24
C9	RET

RAND USR 30000.

Hvis du kigger nøje efter, vil du se, at der bliver udskrevet en række heksadecimale tal, som er lig dem, du selv lige har indtastet. Her følger forklaringen:

Vi starter med at LOAD'e HL med startadressen for rutinen. B LOAD'es med rutinens længde i bytes. Med LD A,(HL) henter vi første byte ind i A-registeret. Den gemmer vi til senere brug med PUSH. Derefter nulstiller vi bit 0, 1, 2 og 3 med AND 240, der er AND 11110000b. Du husker vel, at når der sammenlignes med 0 i en AND-instruktion, bliver bit'en nulstillet. Nu har vi de fire højeste bits tilbage. De udgør tilsammen det første heksadecimale ciffer. Dette ciffer roterer vi over til de 4 første bits. Vi ved,

at tegnene 0 til F har koderne 28 til 43, og dette benytter vi os af nu. Det første ciffer ligger nu som et tal mellem 0 og F heksadecimalt, mellem 0 og 15 decimalt. Tillægger vi denne værdi 28, får vi præcis den kode, som det pågældende tegn har. Den udskriver vi så med RST 10h. Nu tager vi det oprindelige tal tilbage og bruger samme fremgangsmåde, dog uden rotation: de står jo som bit 0 til 3. Sluttelig laver vi et par mellemrum inden næste heks-kode findes. Hvis du ønsker at udskrive andre bytes, evt. i ROM, kan du blot ændre HL-registerets indhold til startadressen. Fordi vi bruger B-registeret, kan du dog kun udskrive 256 tal – men skærmen har jo også sin begrænsning!!

Kapitel 17

Vi vil nu kigge lidt på de instruktioner, som vi endnu ikke har benyttet. Det bliver en kursorisk gennemgang af hver instruktion, idet de simpelt hen er så komplicerede, at vi ikke ønsker at komme nærmere ind på dem. En del af instruktionerne bør man holde sig langt fra, medmindre man er en rigtig skrap programmør.

Den første af instruktionerne er IN-instruktionen. Med denne trækker man indholdet fra én af ZX 81's porte ind i et register. Man kan bl.a. bruge det til kontrol af tastaturet, men vi vil vende tilbage til en nemmere metode til dette end brug af IN-ordren.

Instruktionerne er:

Heks-kode	Mnemonic	
DB	IN A,(n)	n er en port
ED78	IN A,(C)	(C) er den port, register C peger på
ED40	IN B,(C)	
ED48	IN C,(C)	
ED50	IN D,(C)	
ED58	IN E,(C)	
ED60	IN H,(C)	
ED68	IN L,(C)	

En helt speciel IN-instruktion hedder IN F,(C). Ved udførelse af denne instruktion vil kun flagene blive påvirket. Indholdet på porten specificeret i register C vil altså ikke blive hentet ind, men blot aflæst. Indholdet lægges ikke i flag-registeret, som mnemonic'en kan få en til at tro; flagene påvirkes blot, som hvis indholdet var hentet ind.

Heks-kode	Mnemonic
ED70	IN F,(C)

OUT-instruktionen er lige modsat. Her føres en værdi ud til en port. Disse instruktioner er

Heks-kode	Mnemonic
D3	OUT (n),A
ED79	OUT (C),A
ED41	OUT (C),B
ED49	OUT (C),C
ED51	OUT (C),D
ED59	OUT (C),E
ED61	OUT (C),H
ED69	OUT (C),L

Der findes også blok-instruktioner, der opererer med porte. Instruktionen INI laver dette lille program:

```
IN (HL),(C)
INC HL
DEC B
```

IN (HL),(C) findes ikke som selvstændig instruktion. Indholdet på port nr. C lægges på den adresse, HL peger på. Herefter incrementeres HL og B decrementeres. Naturligvis kan løkken gøres repeterende – navnet på instruktionen er INIR. Den repeterer indtil B er nul. IND og INDR findes også. Her decrementeres HL i stedet.

Heks-kode	Mnemonic
EDA2	INI
EDB2	INIR
EDAA	IND
EDBA	INDR

Du kan nok regne ud, at også OUTI, OTIR, OUTD og OTDR findes. OUTI udfører dette program:


```
OUT (C),(HL)
INC HL
DEC B
```

Ved OUTD decrementeres HL. OTIR og OTDR repeterer til B er nul. Koderne er:

Heks-kode	Mnemonic
EDA3	OUTI
EDB3	OTIR
EDAB	OUTD
EDBB	OTDR

Den næste instruktion er HALT, som har kode 118, der jo er Newline. HALT indleder en uendelig løkke NOP, og hermed menes, at programudførelsen kører fast. Der er kun en udvej: Sluk og begynd forfra.

Instruktionen DI forhindrer Z80 i at reagere på interrupts, mens EI (Enable Interrupts) ophæver virkningen af DI (Disable Interrupts), og Z80 vil atter reagere på interrupts. Koderne er:

Heks-kode	Mnemonic
76	HALT
FB	EI
F3	DI

De sidste fem instruktioner er lige så slemme som EI og DI. RETI og RETN er i familie med RET-instruktionen, men begge har med interrupts at gøre. IM 0, IM 1 og IM 2 er tre former for INTERRUPT-MODES, men disse tre og de forrige instruktioner bør du holde dig fra.

For god ordens skyld har du koderne her:

Heks-kode	Mnemonic
ED45	RETN
ED4D	RETI
ED46	IM 0

ED56	IM 1
ED5E	IM 2

Nu har vi omtalt SAMTLIGE instruktioner i Z80-instruktionssættet. Det er op til dig selv at lave programmer. Lav nogle rutiner, du kan kalde fra et BASIC-program. Senere vil du kunne skrive programmer i ren maskinkode. Du skal eksperimentere med de forskellige instruktioner – men husk: Du skal ikke bruge de sidste 8 instruktioner. En naturlig reaktion ved udførelse af en af dem vil være crash. I det hele taget bør du gemme alle instruktionerne i dette kapitel, til du får mere erfaring og viden.

Vi er – som sagt – nu færdige med gennemgangen af instruktionerne. I de kommende kapitler vil vi fortælle om brugen af maskinkode. Vi bidrager med nogle gode tips til videre programmering. Vi vil bl.a. omtale keyboardscanning, hvor tastaturet behandles, ROM-rutiner, brug af beregnerstakken m.m. Der vil komme seks programmer, deriblandt et komplet Space Invaders, som du så selv kan udvide med detaljer.

Kapitel 18

Vi vil i dette kapitel gennemgå samtlige instruktioner. For ikke at drukne i forkortelser og uoverskuelige notationer bringer vi ikke direkte mnemonics, men blot en omtale af instruktionernes virkemåde.

ADC: Der findes ADC både i 8 bits- og 16 bits-form, og i begge tilfælde lægges to værdier sammen. Ved 8 bits adderes på A, mens HL benyttes ved 16 bits-addition. Resultatet tillægges én, hvis C-flaget er sat.

ADD: Den virker som ADC, men værdien af C-flaget tillægges ikke. Endvidere kan 16 bits-additionen også foregå på HL-, IX- eller IY-registeret.

AND: De enkelte bits i A sammenlignes med et register eller et tal. Resultatet af hver enkelt bit-sammenligning bliver 1, hvis begge de sammenlignede bits er 1. Resultatet lægges i A.

BIT: En bit i et register testes, og Z-flaget antager den modsatte værdi af den testede bit.

CALL: En underrutine kaldes, og der vendes tilbage med RET. Instruktionen kan gøres betinget af 4 flags status. Det er C-, Z-, P/V- og S-flaget.

CCF: C-flagets værdi vendes. Hvis det er sat, resættes det, og omvendt.

CP: A sammenlignes med et register eller et tal. Ingen registre påvirkes. Er de lig hinanden, sættes Z-flaget, og C-flaget resættes. Er A størst, resættes begge. Er A mindst, sættes C-flaget, mens Z-flaget resættes.

CPD: Virker som CP (HL), men efterfulgt af DEC HL og DEC BC. P/V-flaget resættes, hvis BC bliver nul. Z-flaget resættes, hvis A er lig (HL).

CPDR: Virker som CPD, men der repeteres til BC er nul, medmindre værdien i A findes i den gennemsøgte blok af bytes. Flagene påvirkes på samme måde som ved CPD.

CPI: Virker som CPD, men HL incrementeres.

CPIR: Virker som CPDR, men HL incrementeres.

CPL: Hver enkelt bit i A vendes.

DAA: Gør et resultat efter addition, subtraktion o.lign. gyldigt i BCD-aritmetik.

DEC: Værdien i et register (8 eller 16 bits) gøres én mindre.

DI: Forhindrer Z80 i at reagere på et interrupt, indtil EI mødes.

DJNZ: B-registeret decrementeres, og der hoppes et angivet antal bytes mellem −128 og 127 frem (tilbage), hvis B ikke er nul.

EI: Z80 vil reagere på et interrupt (ophæver DI).

EX: Denne ombytter enten toppen af stakken med HL, IX eller IY, eller DE ombyttes med HL. Endelig kan AF ombyttes med AF'.

EXX: Ombytter de alternative registre BC', DE' og HL' med de fungerende af samme navn.

HALT: Starter en uendelig løkke af NOP-instruktioner.

IM: Angiver interrupt-tilstand (0, 1 eller 2).

IN: Instruktionen tager indholdet på en given port ind i A eller indholdet fra en port specificeret i C ind i et register (8 bits).

IND: Udfører IN (HL),(C), hvor (C) er indholdet på den port, register C peger på, men efterfulgt af DEC B og DEC HL.

INDR: Virker som IND, men repeterer til B er nul.

INI: Virker som IND, men HL incrementeres.

INIR: Virker som INDR, men HL incrementeres.

INC: Værdien i et 8 eller 16 bits-register forøges med én.

JP: Springer til en adresse mellem 0 og 65535, der er angivet. Kan også springe til adressen i HL, IX eller IY. Instruktionen kan gøres betinget af flagene C, Z, P/V og S's status.

JR: Der hoppes et angivet antal bytes mellem −128 og 127 frem (tilbage). Instruktionen kan gøres betinget af flagene C og Z.

LD: Bruges til at give registre (alle på nær F-registeret) værdi, 8 eller 16 bits. En værdi kan også lægges på en adresse, eller værdien på en adresse kan lægges i et register.

LDD: Udfører LD (DE),(HL) efterfulgt af DEC HL, DEC DE og DEC BC.

LDDR: Virker som LDD, men repeterer, indtil BC er nul.

LDI: Udfører det samme som LDD, men DE og HL incrementeres.

LDIR: Virker som LDDR, men DE og HL incrementeres.

NEG: Hver enkelt bit i A komplementeres, og der tillægges én.

NOP: Udfører ingenting.

OR: De enkelte bits i A sammenlignes med et register eller et tal. Resultatet af hver enkelt bit-sammenligning bliver 1, hvis blot én af de sammenlignede bits er 1. Resultatet lægges i A.

OUT: Indholdet i A lægges ud på en given port, eller indholdet fra et register (8 bits) lægges på den port, som register C peger på.

OUTD: Udfører OUT (C),(HL) efterfulgt af DEC HL og DEC B.

OTDR: Virker som OUTD, men repeterer, til B er nul.

OUTI: Virker som OUTD, men HL incrementeres.

OTIR:Virker som OTDR, men HL incrementeres.

POP: Trækker værdien øverst på stakken ned i et 16 bits-register.

PUSH: Lægger en værdi fra et 16 bits-register op på stakken.

RES: Resætter en bit i et 8 bits-register.

RET: Bruges til at returnere fra en underrutine. Der kan vendes tilbage til BASIC med denne instruktion, der kan gøres betinget af fire flags status, nemlig C, Z, P/V og S.

RETI: Der returneres fra en interrupt-underrutine.

RETN: Der returneres fra en underrutine og interrupt-status inden kald genetableres.

RL: Hver bit i et 8 bits-register flyttes én til venstre. Bit 7 flyttes til C-flaget, og bit 0 tager C-flagets værdi.

RLA: Virker som RL A, men er en byte kortere.

RLC: Hver enkelt bit i et 8 bits-register flyttes én til venstre. Bit 7 flyttes til C-flaget og til bit 0.

RLCA: Virker som RLC A, men en byte kortere.

RLD: Denne må ikke forveksles med RL D. Denne instruktion flytter de fire første bits i A til de fire første i (HL), der flyttes til de fire sidste bits (bit 4-7) i (HL), der flyttes til de fire første bits i A.

RR: Hver enkelt bit i et 8 bits-register flyttes én til højre. Bit 0 går til C-flaget, og C-flaget til bit 7.

RRA: Virker som RR A, men en byte kortere.

RRC: Flytter hver bit i et 8 bits-register én til højre. Bit 0 går til C-flaget og bit 7.

RRCA: Virker som RRC A, men en byte kortere.

RRD: Undgå forveksling med RR D. Her flyttes bit 0 til 3 i A til

bit 4 til 7 i (HL), der flyttes til de fire første bits i (HL), der flyttes til de fire første bits i A.

RST: Kalder forskellige ROM-rutiner. Startadressen er fastlagt af Zilog. Der findes 8 i alt.

SBC: Der findes både 8 bits- og 16 bits-SBC. Ved 8 bits-subtraktion subtraheres på A, mens HL benyttes ved 16 bits. Der fratrækkes yderligere én, hvis C-flaget er sat.

SET: Sætter en bit i et 8 bits-register.

SLA: Hver enkelt bit i et 8 bits-register flyttes til venstre. Bit 7 går til C-flaget, og bit 0 resættes.

SRA: Hver enkelt bit i et 8 bits-register roteres til højre. Bit 0 går til C-flaget, mens bit 7 ikke berøres.

SRL: Hver bit i et 8 bits-register flyttes til højre. Bit 0 går til C-flaget, og bit 7 resættes.

SUB: Virker som SBC, men der findes ikke 16 bits-subtraktion. Endvidere fratrækkes værdien af C-flaget ikke.

XOR: De enkelte bits i A sammenlignes med et register eller et tal. Resultatet af hver enkelt bit-sammenligning bliver 1, hvis én og kun én af de sammenlignede bits er 1. Resultatet lægges i A.

Atter minder vi om, at oversigten kun er summarisk. Ikke alle tilfælde er nævnt. Yderligere information kan du naturligvis finde i den nøjere beskrivelse af instruktionerne, som vi netop har givet.

Kapitel 19

I dette kapitel vil vi starte på de mange tips, som vi bringer. Først viser vi, hvordan du kan bruge tastaturet, (keyboardet) i dine rutiner.

Der er flere forskellige metoder. Vi kunne bl.a. bruge IN-instruktionen og aflæse en bestemt port, men vi har fundet det mere naturligt at bruge den rutine i ROM, som ZX 81 (og Lambda) bruger, når du benytter tastaturet. Først skal vi dog se lidt nærmere på selve tastaturet (keyboardet).

Lad os betragte tastaturet horisontalt. Vi inddeler det i 8 dele, og hver sektion får et nummer.

sektion 3	1	2	3	4	5	6	7	8	9	0	sektion 4
sektion 2	Q	W	E	R	T	Y	U	I	O	P	sektion 5
sektion 1	A	S	D	F	G	H	J	K	L	N/L	sektion 6
sektion 0		Z	X	C	V	B	N	M	.	SPC	sektion 7

Når vi kalder keyboardscanningen, som ROM-rutinen, der tager sig af tastaturet, hedder, vil der komme et resultat i HL-registeret. L-registeret tager sig af resultatet fra ovenstående illustration. Hvis en tast fra sektion 0, Z f.eks., nedtrykkes, bliver resultatet 11111110b = 254, idet bit 0 resættes. Trykkes der på H, bliver resultatet i register L 10111111b = 191. Reglen er simpelt hen: Den bit, der svarer til den sektion, hvori en tast er nedtrykket, resættes. Hvis ingen taster er nedtrykket, vil L blive 11111111b = 255. Der bliver alle bits sat, som det ses. Trykkes f.eks. både Z og H, bliver resultatet 10111110b = 190, og nedtrykkes en tast fra hver sektion, bliver L-registerets indhold nul. Shift-tasten vil, som det ses på tegningen, ikke påvirke L-registeret.

Imidlertid kan vi også betragte tastaturet vertikalt, og det ser således ud:

	SH		2 W S Z	3 E D X	4 R F C	5 T G V	6 Y H B	7 U J N	8 I K M	9 O L .	Ø P N/L SPC
sekt.-nr.	0	1	2	3	4	5	5	4	3	2	1

sektionsnummer

Her virker det på samme måde, men H-registeret påvirkes i stedet. Trykkes A ned, resættes bit 1 i H-registeret, og resultatet bliver 11111101b = 253.

Som det ses, kan bit 6 og 7 ikke resættes, og H's mindste værdi er derfor 192 (128+64). SHIFT-tasten har her sin egen sektion i modsætning til den horisontale betragtning, hvor den ikke er med. Trykkes SHIFT, vil H-registerets bit 0 resættes, og derfor vil værdien i H for en SHIFT'et tast, være én mindre end for tasten alene.

Eksempel: Hvis du trykker på 5, vil bit 5 i H være resat, og bit 3 i L vil også være resat. Det er 11011111b * 256 + 11110111b = 223 * 256 + 247 = 57335.

Som du forhåbentlig kan se nu, så har hver tast sin egen værdi. Vi vil nu påvise, at HL-registeret får værdien 57335, når du trykker på 5.

Heks-kode	Mnemonic	Forklaring
CDBB02	CALL 699	Kald keyboardscanningen
(CD3D03 – CALL 829)		
44	LD B,H	Kopiér HL
4D	LD C,L	i BC
24	INC H	Læg én til H
28F8	JR Z, –8	Gå tilbage, hvis ingen taster er nedtrykket (H=255)
C9	RET	Returner til BASIC

Vi kalder først ROM-rutinen, dernæst kopierer vi HL i BC, så vi er klar til udskrift. Er der ikke trykket på nogen tast, er H 255. Vi lægger én til, og H bliver 0, hvis ingen taster er påvirket. Der hoppes tilbage, hvis dette er tilfældet. Ved at kalde med PRINT USR 30000, får vi skrevet værdien i HL ud. Resultatet, hvis du ellers

har trykket på 5, er 57335. Prøv at trykke på andre taster, evt. flere, hvis du kan trykke præcist samtidigt – og det kan være svært!

Keyboardscanningen ligger som skrevet på adresse 699 (829) i ROM. Der er dog en ulempe, og det er, at den både bruger A-, BC-, DE- og HL-registeret. Skal du bruge et indhold af disse registre efter scanningen, må du altså først PUSH'e dem. Efter scanningen kan du så POP'e dem tilbage, når du har behandlet resultatet.

Bemærk i øvrigt, at vi i programmet har placeret overførslen til BC før JR Z-ordren. Det ville være mere nærliggende først at overføre *efter*, at der er trykket på en tast, men det vil koste en byte, da H så skal decrementeres!

Men der er andre muligheder end CALL 699. Der er en systemvariabel, som styres af keyscan-rutinen. Denne variabel er på adresse 16421 og 16422. Det hele kan altså ordnes med LD HL,(16421). Vores program kunne altså i stedet se således ud:

Heks-kode	Mnemonic
ED4B2540	LD BC,(16421)
04	INC B
28F9	JR Z, –7
05	DEC B
C9	RET

PRINT USR 30000.

I ROM er inkluderet flere rutiner til brug ved keyboardscanning. Fra adresse 126 (123) ligger der tegn på en lang række, og de er placeret efter deres sektionsplads. De ligger således: ZXCVASD FGQWERT12345Ø9876POIUY N/L LKJH SPC .MNB plus alle de tegn og kommandoer, som på ZX 81 er skrevet med rødt på tastaturet (dekan alle kaldes med SHIFT). Den rutine, som gør brug af denne tabel, ligger på adresse 1981 (2393). Den skal du bruge således: Når du kalder 699 (829), kommer en kodeværdi, som beskrevet, for den tast, du har nedtrykket, i register HL. Denne værdi overføres til BC. Derefter kaldes rutine 1981 (2398). Når den er færdig, vil der ligge et tal i HL. Dette tal peger netop på det pågældende tegn i tabellen, så hvis du

skriver LD A,(HL) og RST 10h, vil du få udskrevet det tegn, som du trykkede på.

Hvis du nedtrykker to taster samtidig – og SHIFT ikke er den ene – ved ZX 81 (Lambda ligeledes) ikke, hvilket af tegnene den skal skrive. I stedet skriver den et mellemrum.

Du må dog ikke kalde rutinen på 1981 (2398) uden at have nedtrykket en tast. Da kan ZX 81 jo ikke skrive noget, og det er *ikke* ensbetydende med SPACE!!!

Heks-kode	Mnemonic
START	
CDBB02	CALL 699
(CD3D03 – CALL 829)	
44	LD B,H
4D	LD C,L
2C	INC L
28F8	JR Z, START
CDBD07	CALL 1981
(CD5E09 – CALL 2398)	
7E	LD A,(HL)
D7	RST 10h
111027	LD DE,10000
1B	DEC DE
7B	LD A,E
B2	OR D
20FB	JR NZ, –5
CD460F	CALL 3910
(CD3B11 – CALL 4411)	
D0	RET NC
18E5	JR START

Vi starter med at kalde keyboardscanningen. Herefter kopieres HL i BC, som vores nye rutine jo kræver. Hvis der ikke er aktiveret nogen tast, SHIFT undtaget, vil register L være 255. Vi husker, at SHIFT kun resætter bit 0,H. Hvis ingen er aktiveret, vil L være nul efter INC L. Derefter startes evt. forfra. Bemærk, at der alligevel startes forfra, hvis SHIFT er trykket ned. Hvis en tast er aktiveret, fortsættes der. Den nye rutine kaldes nu, og A

LOAD'es med koden for det nedtrykkede tegn. Det udskrives med RST 10h. Derefter er der pause, og vi benytter den gamle undersøgelse for, om BREAK er aktiveret. Hvis det er tilfældet, returneres til BASIC. JR starter rutinen om igen.

Inden du kører programmet (med USR 30000), husker du, at kun én tast må nedtrykkes for ikke at give mellemrum. Hvis du trykker N/L, skiftes der linie. Hvis du ønsker at skrive f.eks. »+«, trykker du SHIFT, og holder den nede. Tryk nu + (K på ZX 81 – L på Lambda), og sejren skulle være hjemme... Du må ikke trykke SHIFT og f.eks. E ned samtidig. Så vil A jo indeholde koden for STEP (TAN på Lambda), og RST 10h udskriver kun displaybare tegn, hvortil dette ikke hører. Maskinen vil crashe! Vi giver senere i bogen – under programmer – masser af eksempler på keyboardrutinen.

Det var slut på dette kapitel. Har du lyst til at prøve en videreførelse af ovenstående program, så kommer det under programeksemplerne. Prøv nu selv at lave et program, så du kan styre et A rundt på skærmen – uden at det kører udenfor. Du kan evt. lave en sort kant om hele skærmen. Så kan du i programmet undersøge, om A'et vil komme ud i kanten. Hvis det er tilfældet, skal kørselsretningen ikke accepteres, naturligvis. Pauser er nødvendige for, at A'et ikke farer fra den ene ende til den anden på brøkdele af et sekund.

Find selv på flere ideer. Prøv at udvikle programmerne – du har sikkert også helt private ideer til programeksempler. Brug dem.

Kapitel 20

I dette kapitel vil vi gennemgå en hel masse rutiner i ROM, som kan være nyttige at bruge i egne programmer. Mange af dem vil også gøre livet en del lettere at leve, hvis man er maskinprogrammør vel at mærke.

Den første har med skærmbilledet at gøre, og den er uhyre nyttig. Hvis du nu ville skrive en hel sætning på skærmen, tror du måske, at du skal LOAD'e hvert enkelt tegn ind i register A og så eksekvere RST 10h. Tænk, hvis der skulle skrives 100 tegn. Så måtte der bruges 300 bytes, medmindre du lavede en løkkestruktur med de tegn, der skulle udskrives i en blok for sig. Alligevel ville det være meget besværligt. Til alt held findes der en rutine i ROM, som ordner den slags noget nemmere. Du placerer bare dine tegn et sted i RAM. For at kalde rutinen LOAD'er man DE med startadressen for sine tegn, der altså skal være samlet som en blok. Har du 100 tegn fra adresse 30000 til 30099, skal DE LOAD'es med 30000. Desuden skal BC LOAD'es med det antal tegn, der skal udskrives, altså LD BC,100 i ovenstående tilfælde. Derefter kalder man bare rutinen på adresse 0B6Bh (0CF6h), der er 2923 (3318) decimalt. Hvis der i tegnblokken står et tegn, som ikke er displaybart, vil rutinen også selv sørge for at ordne det. Nu kan vi pludselig lave et program, som udskriver HELE tegnsættet. Inden vi laver det, vil vi gøre opmærksom på, at registrene ikke vender tilbage med det, vi LOAD'ede dem med. DE vil returnere med BC-værdien + DE-værdien. BC vil være 65535. Men nu til programmet: Vi lader adresse 30000 være det tegn, vi er nået til.

Heks-kode	Mnemonic	Forklaring
00	DEFB	Nummeret på det tegn, vi er nået til – nul til start
113075	LD DE,30000	Startadresse for tegnene
010100	LD BC,1	Antallet af tegn, der skal udskrives
CD6B0B	CALL 2923	Udskriv tegnet
(CDF60C – CALL 3318)		
213075	LD HL,30000	Her står nummeret på tegnet
7E	LD A,(HL)	Læg nummeret i A
FEFF	CP 255	Er det 255 (det sidste)?
C8	RET Z	Så returner til BASIC
34	INC (HL)	Forøg nummeret med én
18ED	JR –19	Hop tilbage

Vi starter med at LOAD'e DE med startadressen (30000), og BC med længden (1). Der er jo kun et tegn ad gangen. Det udskrives. Så LOAD'es HL med 30000, hvor tegnet står. A LOAD'es med tegnets kode og sammenlignes med 255. RET Z returnerer, hvis A er 255, og der dermed ikke er flere tegn i sættet. Hvis der er, gøres (HL) én større, og byte 30000 indeholder nu næste tegn. Der startes om på ny med JR –19.

Programmet kan startes med RAND USR 30000. Husk, at 30000 jo kun er DATA. Der vil ikke ske noget ved det, da PEEK 30000 jo er nul, hvilket er instruktionen NOP. Men lad være med at gøre det. Det bliver måske bare en vane at gøre det. En dag står der noget, som vil få ZX 81 til at crashe, og så er indtastningen spildt.

Rutinen finder i øvrigt også selv ud af, om skærmen er fyldt.

Prøv for resten RAND USR 2919 (3314), altså et ROM-kald.

Næste rutine er PRINT AT-rutinen. Som navnet siger, kan man selv bestemme, hvor på skærmen der skal skrives. Inden den kaldes, skal B-registeret indeholde linienummer, og C-registeret kolonnenummer. Hvis du f.eks. ønsker at skrive på 10,18 svarende til PRINT AT 10,18;... gøres følgende:

```
LD B,10
LD C,18
CALL 2293
  (CALL 2708)
```

Da B og C jo udgør BC-registeret, kan programmet trækkes sammen. BC indeholder så 256 x 10 + 18 = 2578, altså

```
LD BC,2578
CALL 2293
  (CALL 2708)
```

Igen returneres registrene med andre værdier end dem, vi sendte ud.

Eksempel på et program:

Heks-kode	Mnemonic	Forklaring
010805	LD BC,1288	Sæt printposition til
CDF508	CALL 2293	AT 5,8
(CD940A – CALL 2708)		
3E89	LD A,137	Giv A værdien 137
D7	RST 10h	Udskriv tegnet med koden 137
C9	RET	Returner til BASIC

Som du ser ud fra heks-koderne, indeholder C 8 og B 5, så programmet udskriver altså CHR$ 137 »AT 5,8«.

Nu når vi har fat i RST 10h, vil vi nævne, at du måske en dag støder ind i CALL 0808h (09A9h). Denne rutine i ROM gør præcis det samme som RST 10h, men er to bytes længere, og den lægger desuden andre værdier i registrene. I egne programmer er RST 10h at foretrække, men prøv for sjovs skyld at ombytte denne instruktion med CALL 0808h (09A9h) i ovenstående program. Du kan forhåbentlig selv finde ud af, at heks-koden er CD0808 (CDA909). Samme resultat, ikke?

Du kan måske også få brug for at PLOT'te, så derfor ligger der en rutine i ROM på adresse 0BB6h (0D3Dh) – det er 2998 (3389) i decimal notation. Før du kalder rutinen, skal B-registeret

indeholde y-koordinaten, dvs. antallet af punkter fra skærmens nederste kant i opadgående retning. Naturligvis skal C-registeret indeholde X-koordinaten.

SCROLL har vi omtalt, men for en god ordens skyld skal vi sige, at den ligger på adresse 0C0Eh (0BB1h).

Hvis du får brug for at rydde skærmen, så ligger CLS på adresse 0A2Ah (OBDOh).

En anden nyttig rutine ligger i ROM på adresse 2712 (3117). Denne rutine udskriver indholdet af BC-registeret som decimaltal. Den kan dog ikke udskrive værdier, hvis de er over 9999. Rutinen er meget brugbar, da man ikke på anden måde, som vi kender, kan få udskrevet decimaltal på en nem måde, og decimaltal foretrækker vi mennesker. Når rutinen kaldes, vil de værdier, der er i registrene, ændres.

Eksempel:

Heks-kode	Mnemonic	Adresse	Forklaring
010000	LD BC,0	30000	Nulstil vor tæller
C5	PUSH BC	30003	Læg den op på stakken
CD980A	CALL 2712	30004	Udskriv tællerens værdi
(CD2D0C – CALL 3117)			
CD0E0C	CALL 3086	30007	Scroll
(CDB10B – CALL 2993)			
C1	POP BC	30010	Hiv tælleren ned
03	INC BC	30011	Læg én til
CD460F	CALL 3910	30012	Er BREAK aktiveret,
(CD3B11 – CALL 4411)			
D0	RET NC	30015	så returner til BASIC
C33375	JP 30003	30016	Hop tilbage

Det første, vi gør, er at lægge 0 i register BC og derefter lægge værdien op på stakken. Det gør vi for at huske, hvad BC er. Rutinen i ROM bruger jo BC-registeret! Den kaldes nu, og vi scroller med CALL 3086 (2993). BC hentes ned fra stakken igen og incrementeres. Så returneres til BASIC, hvis BREAK er nedtrykket. Derefter vendes tilbage til 30003, hvor PUSH BC står.

En variant kunne være at få scrollet dele af et hukommelsesområde op f.eks. udskrift af hele ROM ved at starte i adresse 0,

bede den om at PEEK'e denne, udskrive den, scrolle den, og dernæst tage næste adresse. Lav det selv!

Du får naturligvis brug for SLOW og FAST i dine rutiner. De ligger på hhv. 0F2Bh (0281h) og 0F23h (036Ch). Du får i hvert fald brug for FAST, hvis du ønsker at SAVE'e og LOAD'e. Hvis man kalder SAVE- eller LOAD-rutinen i ROM direkte *skal* ZX 81 (Lambda) køre i FAST, ellers sker der ingenting. Hvis du har SAVE'et et program i maskinkode, kan du LOAD'e det tilbage i BASIC, men det omvendte kan ikke lade sig gøre. Selvfølgelig kan man LOAD'e et program i maskinkode, som er SAVE'et i maskinkode.

Hvis du vil SAVE'e, skal du altså udføre dette lille program:

```
CALL FAST
CALL SAVE
CALL SLOW
```

SAVE-rutinen ligger på 02FBh (0388h), og LOAD-rutinen på 0347h (030Fh). SAVE-rutinens startadresse er altså 763 (904), men det kan gøres lidt kortere. Hvis du i BASIC skriver FAST, og derefter RAND USR 787 (928), vil du opnå SAVE uden de fem sekunders pause, som du kender.

Endnu en gang bør du eksperimentere videre selv med de tips, vi har givet dig. Du kan jo evt. selv prøve at finde flere rutiner i ROM ved at gå på jagt i den. Hvis du har en printer, vil en rutine svarende til COPY være velegnet.

Til jagten vil en såkaldt disassembler være hensigtsmæssig. En disassembler oversætter fra heksadecimale tal (egentlig binære tal) til assembler, altså den modsatte vej af en assembler. Her får du lidt råd med på vejen, hvis du vil lave en disassembler, men det er kompliceret. Hør blot:

For at kunne gå på jagt i ROM er det nødvendigt at lave et program, som vil være dig behjælpelig med opgaven. En mulighed er at lave den føromtalte lille rutine, der udskriver heksadecimale koder. Dette er dog en meget tidkrævende løsning, fordi du er nødt til at slå op i appendiks A for at oversætte det til en mnemonic, vi kan forstå. Der er brug for et anderledes program,

som til gengæld er uhyre kraftfuldt. Vi tænker på en udskrift som f.eks.

```
 0 OUT (255),A
 2 LD BC,32767
 5 JP 971
 8 LD HL,(16406)
11 LD (16408),HL
```

Det er et sådant program, som alle maskinkodeprogrammører drømmer om at eje. Drømmen kan blive en realitet, hvis du har mod på udfordringer. Vi vil i det følgende give dig opskriften på at lave et sådant program. Dette kaldes en disassembler i modsætning til en assembler, som oversætter mnemonics til heksadecimale (binære) tal. Vi vil ikke selv liste vores version af programmet, men simpelt hen overlade det til dig selv at lave programmet, og vi bør nok allerede nu påpege, at det bliver en svær øvelse, hvis du har store krav om effektivitet og knaphed.

Programmet kan forholdsvis nemt laves ved at lave en tabel i programmet, som indeholder *alle* mnemonics, og så lade ZX 81 finde instruktionen i denne blok og herefter udskrive den. Men dette ville være et enormt tastearbejde, da alene alle disse data ville fylde ret meget i RAM. Det bliver faktisk omkring 4 K. Dertil kommer så selve programmet, som skal finde ud af, hvor i tabellen instruktionen står m.m.. Den metode vil unægtelig være meget omfangsrig, og det vil nærmest være at gøre grin med Z80A-processoren. Et sådant lidet kraftfuldt program totalt uden elegance vil være under den tilfredsstillende standard.

Den metode, som vi vil beskrive for dig, kan ved hjælp af et godt program (det er op til dig selv, og din kunnen ikke mindst) skrives på omtrent 1 K. Men du bør huske: Det kan blive meget svært!

Taber du tråden på et tidspunkt i den hastige gennemgang, så blad tilbage, og vær sikker på, at du forstår det.

For at lave programmet, skal vi bruge følgende konventioner:

R(0) = B, R(1) = C, R(2) = D, R(3) = E, R(4) = H, R(5) = L,
R(6) = X, R(7) = A

S(0) = BC, S(1) = DE, S(2)=Y, S(3)=SP

Q(0) = BC, Q(1) = DE, Q(2) = Y, Q(3) = AF

N(0) = 0, N(1) = 1, N(2) = 2, N(3) = 3, N(4) = 4, N(5) = 5,
N(6) = 6, N(7) = 7

C(0) = NZ, C(1) = Z, C(2) = NC, C(3) = C, C(4) = P0, C(5) = PE,
C(6) = P, C(7) = M

X(0) = ADD A,, X(1) = ADC A,, X(2) = SUB, X(3) = SBC A, ,
X(4) = AND, X(5) = XOR, X(6) = OR, X(7) = CP

Så skal den byte, som skal disassembles, omskrives til binær notation. Del dette tal ind i 3 dele: F, G og H. F består af bittene 7 og 6, G af bittene 5, 4 og 3, og H består af bittene 2, 1 og 0. Dernæst deles G's 3 bits ind i J og K, hvor J består af G's bit 2 og 1, og K af G's bit 0.

Vi tager et eksempel:

Hvis AAh skal deles op: AA = 10/101/010, bliver F=2, G=5 og H=2. G er 10/1 binært, og delt op bliver J 10b – og det er 2 – og K 1.

Derfor kan F, G, H, J og K antage følgende værdier:

F kan være 0, 1, 2, eller 3
G kan være 0, 1, 2, 3, 4, 5, 6, eller 7
H kan være 0, 1, 2, 3, 4, 5, 6, eller 7
J kan være 0, 1, 2, eller 3
K kan være 0 eller 1

Disse variable bruges i resten af forslaget.

Start:

Næste byte, som skal disassembles, findes, og her sker der en opdeling i 4 kategorier:

1) Hvis byten er 76h, er instruktionen HALT
2) Hvis byten er CBh, så hop til afsnittet om CBh
3) Hvis byten er EDh, så hop til afsnittet om EDh

4) Hvis byten er DDh eller FDh, så begynd testen forfra ved START

Endelig kan det være en helt femte talværdi, og det er tegn på, at det er en »normal« instruktion.

Disse skelner vi således:

Hvis F er nul, så
  hvis H er nul, så
    hvis G er større end 3, er koden JR C(G–4),d
    hvis G er mindre end 4, så brug den G'ende værdi i denne tabel:NOP/EX AF,AF'/DJNZ d/JR d

  hvis H er en, så
    hvis K er nul, er koden LD S(J),nn
    hvis K er en, er koden ADD Y,S(J)

  hvis H er to, så

    lad koden være LD plus den G'ende værdi i denne tabel: (BC),A / A,(BC) / (DE),A / A,(DE) / (nn),Y / Y,(nn) / (nn),A / A,(nn)

  hvis H er tre, så

    hvis K er nul, er koden INC S(J)
    hvis K er en, er koden DEC S(J)

  hvis H er fire, er koden INC R(G)

  hvis H er fem, er koden DEC R(G)

  hvis H er seks, er koden LD R(G),n

  hvis H er syv, så brug den G'ende værdi i denne tabel: RLCA/RRCA/RLA/RRA/DAA/CPL/SCF/CCF

Hvis F er en, så er koden LD R(G),R(H)

Hvis F er to, så er koden X(G) R(H)

Hvis F er tre, så

hvis H er nul, så er koden RET C(G)

hvis H er en, så

hvis K er nul, er koden POP Q(J)
hvis K er en, så brug den J'ende værdi i denne tabel:
RET/EXX/JP (Y)/LD SP,Y

hvis H er to, er koden JP C(G),nn

hvis H er tre, så brug den G'ende værdi i denne tabel:
JP nn / – / OUT (n),A / IN A,(n) / EX (SP),Y / EX DE,HL / DI / EI

hvis H er fire, så er koden CALL C(G),nn

hvis H er fem, så

hvis K er hul, så er koden PUSH Q(J)
hvis K er en, så er koden CALL nn

hvis H er seks, så er koden X(G) n

hvis H er syv, så lad koden være RST plus den G'ende værdi i denne tabel: 00/08/10/18/20/28/30/38/40/48

Det er de »normale« instruktioner. Hvis vi brugte kategorien med CBh, så bruger vi dette på den efterfølgende byte:

Hvis F er nul, så brug den G'ende værdi i denne tabel:
RLC/RRC/RL/RR/SLA/SRA/–/SRL plus R(H)

Hvis F er en, så er koden BIT N(G),R(H)

Hvis F er to, så er koden RES N(G),R(H)

Hvis F er tre, så er koden SET N(G),R(H)

Hvis vi er under EDh, skal vi undersøge den efterfølgende byte således:

F kan ikke være nul efter en ED

Hvis F er en, så

hvis H er nul, så er koden IN R(G),(C)

hvis H er en, så er koden OUT (C),R(G)

hvis H er to, så

hvis K er nul, så er koden SBC HL,S(J)
hvis K er en, så er koden ADC HL,S(J)

hvis H er tre, så

hvis K er nul, er koden LD (nn),S(J)
hvis K er en, så er koden LD S(J),(nn)

hvis H er fire, er koden NEG

hvis H er fem, så

hvis K er nul, er koden RETN
hvis K er en, er koden RETI

hvis H er seks, så brug den G'ende værdi i denne tabel:
IM 0 / – / IM 1 / IM 2 / – / – / – /–

hvis H er syv, så brug den G'ende værdi i denne tabel:
LD I,A / LD R,A / LD A,I / LD A,R / RRD / RLD / – / –

Hvis F er to, så brug den H'ende værdi i denne tabel: LD / CP / IN / OT / – / – / – / – plus den G'ende værdi i denne tabel: – / – / – / – / I / D / IR / DR

F kan ikke være tre efter en ED.

Nu mangler vi kun at udregne det endelige resultat.

Hvis den første byte i instruktionen var DDh, så erstat hvert Y med IX.

Hvis den første byte i instruktionen var FDh, så erstat hvert Y med IY.

Hvis dette ikke er tilfældet, erstat Y med HL.

Hvis den første byte i instruktionen var DDh, så erstat hvert X med (IX+d).

Hvis den første byte i instruktionen var FDh, så erstat hvert X med (IY+d).

Hvis dette ikke er tilfældet, erstat X med (HL).

Her er d et tal mellem −128 og 127 (2-komplementmetoden), og d er altid placeret som det 3. heks-tal, når det indgår i IX- og IY-instruktioner.

n er et 8 bits-tal, og nn er et 16 bits-tal. Som du ved, står sådanne variable altid til sidst i heks-koden, så disse bytes finder du bare.

Dette var faktisk alt, hvad du skal bruge. Husk, at det første element i tabellerne har nummer nul.

For at gøre det hele lidt klarere, snupper vi lige et par eksempler. Lad os sige, at vi vil starte med at disassemble fra adresse 30000, hvor der først står EDB1...

Da der står EDh, hopper vi selvfølgelig ned til dette afsnit, og

dér står, at vi skal kigge på den næste byte, altså B1h. Det er 10110001b, og derfor bliver tallet delt op således:

1 0 / 1 1 : 0 / 0 0 1

F bliver 2, G bliver 6, H bliver 1, J bliver 3 og K bliver 0. Find under EDh; F lig 2. Der bliver vi bedt om at finde den H'ende værdi i tabellen. Da H er 1, er den H'ende værdi i listen CP, og da G er 6, bliver den samlede instruktion CPIR.

Vi tager endnu et eksempel: F28240h. F2h er ikke noget særtilfælde, så vi deler op binært i 1 1 / 1 1 : 0 / 0 1 0, og vi får værdierne F lig 3, G lig 6, H lig 2, J lig 3 og K lig 0. Vi går til F lig 3 og H lig 2, og dér står, at instruktionen er JP C(G),nn. Da G er 6 bliver det JP C(6),nn, og i vores konventionsliste er C(6)=P, så vi får JP P,nn. nn er de to næste bytes i heks-koden, så det bliver JP P,4082h, eller JP P,16514.

Et tredje eksempel: FD56F0h. Da første byte er FD, startes forfra med næste byte, der indeholder 56h. Det er 01 / 01 : 0 / 110, og værdierne bliver F lig 1, G lig 2, H lig 6, J lig 1 og K lig 0. Vi finder, at instruktionen er LD R(G),R(H), eller med vores værdier LD R(2),R(6). Vi kigger i vores liste, og får det til at betyde LD D,X. For at lave sidste udregning så vi, at hvis den første byte i instruktionen var FDh, erstattes X med (IY+d), og vi får LD D,(IY+d). Næste byte er F0h, og det giver den endelige mnemonic: LD D,(IY−16).

Som sagt vil vi nu overlade det til dig selv at lave programmet. Når du har lavet det, så gem det på bånd, inden du prøver det. Én fejl vil sandsynligvis resultere i et crash. Pas i øvrigt på, hvis du når til f.eks. denne: FDCB0136h, som er RES 6,(IY+1). Hvis du fulgte anvisningen om CBh, skulle du tage byten efter denne, og det er 01h, men da der i instruktionen indgår (IY+d), hvor d jo er 3. del i heks-koden, altså netop 01h, vil det give problemer, da det jo først er næste byte igen, som du skal anvende. Tag nogle passende forholdsregler.

Kapitel 21

Der er en ulempe ved at have maskinkoderutiner over RAMTOP. Der er ingen nærliggende mulighed for at få det ind på bånd. Man kunne dog lave en SAVE-rutine selv. SAVE tager sig af RAM op til E-line. Derfor må vi finde ud af at placere rutinerne et bedre sted, og efter vor mening er en REM-linie den bedste løsning. En REM-linie er en BASIC-linie med kommandoen REM, og det skal vel at mærke være den første. Du laver en linie med et antal tegn – ligegyldigt hvilke – svarende til antallet af heksadecimale tal i maskinkodeprogrammet. En CALL-ordre, der jo fylder 3 bytes, vil derfor også fylde 3 bytes i REM-sætningen. Den første adresse, du kan gemme på, er 16514 (17307), da 16509 (17302) og 16510 (17303) optager linienummer, 16511 (17304) og 16512 (17305) længden af linien og 16513 (17306) indeholder REM-kommandoen.

Derfor ændres heks-loaderens første linie til

```
10 LET X = 16514
  (10 X = 17307)
```

Nu kan du indkøre dine rutiner i REM-sætningen, men sørg for, at der tegn nok. Indtaster du flere heksadecimale tal i heks-loaderen, end der er tegn i linien, vil du POKE på »end of line«-koden, 118, og så vil der opstå problemer.

Et andet råd, vi vil give dig, er, at du kan POKE 16510 (17303) med tallet nul. Herved bliver din REM-sætning til linie nul. 16509 (17302) skal også POKE's til nul, hvis linienummeret overstiger 255, og du kan ikke slette den ved en fejltagelse. Linie 0 kan ikke røres fra BASIC, ikke en gang med

```
0 NEWLINE
```


Hvis listningen af REM-sætningen alene fylder mere end en skærm, vil du nogle gange opleve, at ZX 81 crasher. Det skal selvfølgelig forhindres på en eller anden måde. Du kan gøre det med denne metode: Når du kører heks-loaderen skal de to første heksadecimale tal være 7676 (76FF). 76h er jo et Newline (118), og FFh er koden efter programfilen på Lambda. Når disse står først, vil du ved listning kun opnå følgende udskrift:

```
0 REM
```

Naturligvis er linienummeret kun nul, hvis du har POKE'et dette.

Med denne metode kan andre heller ikke se, hvad programmet indeholder, og hvordan det virker. Men RAND USR 16514 (17307) vil derefter kalde instruktionen HALT... Pas på, at du altid kalder programmets start.

Du kan gemme maskinkoderutiner andetsteds, f.eks. i slutningen af programfilen eller i et array, men vi vil ikke give opskriften på disse metoder, da vi anser den netop anviste for den mest anvendelige. Ved placeringen i slutningen af programfilen vil kaldeadressen ændres ved indtastning af BASIC-linier, og derved skal alle JP- og CALL-instruktioner i rutinen også ændres. Hvis du placerer rutinen i et array, vil RUN eller CLEAR fjerne arrayet. Når rutinen skal kaldes, skal du først identificere variabelfilen – hvor rutinen er gemt – og metoden bliver nemt uoverskuelig og besværlig. Hold dig til REM-sætningen...

*

Her følger nu nogle gode råd til maskinkodeprogrammering.

Tit opstår der fejl i et program, og at finde en eller flere af slagsen kan være svært. Vi vil råde dig til først at undersøge stakfejl. Du skal i de fleste tilfælde have lige mange PUSH og POP-instruktioner. Undersøg dernæst flagsætningen. De betingede hop og kald, JR, JP, CALL og RET, kan være udført p.g.a. et flags status, der er modsat det ønskede. Benytter du ROM-rutiner, skal du i reglen huske at stakke registrene inden og hente dem frem

derefter. Du kan ikke påregne, at registrene har samme værdi efter et kald i ROM.

Ved programmeringen skal du undgå at lave programmerne for indviklede. Du – og helst også andre – skal kunne finde ud af dem, og til dette kan du få glæde af at nedskrive kommentarer på samme måde, som vi har gjort det her i vor gennemgang. Metoden er besværlig, men kan vise sig uhyggelig effektiv, også ved fejlfinding. Så vidt det er muligt, bør du lave underprogrammer, du kan kalde fra en hovedrutine. På den måde kan fejlene lettere findes ved at afprøve en under-rutine ad gangen. Vort Space Invaders, der følger bagest i bogen, er mest konstrueret ud fra dette princip. Brug også JR i stedet for JP, hvor du har mulighed for det.

Husk endelig en ting: At lagre dit nyindtastede maskinkodeprogram på bånd, *inden* du afprøver det. Risikoen for en fejl er stor, selv for erfarne programmører.

Til selve programmeringen kan du blandt andet få brug for at manipulere med C-flaget (ikke mindst fordi 16 bits-SUB ikke findes). Her er gode tips, der også viser, hvordan du kan nulstille A-registeret hurtigere end LD A,0.

C-flaget sættes med SCF
C-flaget resættes med AND A / OR A
C-flaget resættes og Z-flaget sættes med CP A
C-flaget resættes, Z-flaget sættes og A-registeret nulstilles med XOR A / SUB A
C-flaget vendes med CCF

Registrene A, HL, IX og IY kan fordoble deres værdi med ADD A,A – ADD HL,HL – ADD IX,IX – ADD IY,IY.

Kapitel 22

Vi har lovet at fortælle om RST 28h-instruktionen, og det kommer nu.

RST 28h bruges til matematiske beregninger. Hertil bruges yderligere en ROM-rutine, og hele operationen foregår på beregnerstakken, hvorefter resultatet via endnu en ROM-rutine lægges tilbage i et register.

Vi starter med den ROM-rutine, der lægger et tal op på beregnerstakken. Den ligger på adresse 5408 (5912) eller 1520h (1718h). Denne rutine lægger værdien i BC-registeret op på beregnerstakken. Dette er en stak, som ZX 81 bruger til sine beregninger. Modsat vores »almindelige« stak (til PUSH og POP), ligger beregnerstakken »rigtigt«, dvs. tallene, der lægges derop, lægges opad rent adressemæssigt. Når vi har lagt et tal derop, kan vi foretage matematiske beregninger. De nemmeste er naturligvis addition, subtraktion, multiplikation og division, men disse beregninger kræver lige så naturligt to tal, så vi må lægge to tal op, før det bliver udført korrekt. Selve operationen bruger RST 28h, der indikerer beregningen. Efter RST 28h er der et tal, der viser, hvilken funktion vi benytter, og det hele sluttes af med heks-koden 34, der viser, at beregningen er færdig. På adresse 0EA7h (1088h) i ROM – det er 3751 (4232) decimalt – er placeret en rutine, der lægger det beregnede tal tilbage i BC.

Måske lyder det lidt indviklet, men vi tager et eksempel:

Heks-kode	**Mnemonic**	**Forklaring**
010D00	LD BC,13	BC får værdien 13
CD2015	CALL 5408	Tallet lægges op på beregnerstakken
(CD1817 – CALL 5912)		
011B00	LD BC,27	BC får værdien 27

CD2015	CALL 5408	Op på beregnerstakken med det
(CD1817 – CALL 5912)		
EF	RST 28h	Vi gør klar til beregning
04	DEFB	Vi multiplicerer
34	DEFB	Vi afslutter beregningen
CDA70E	CALL 3751	Vi lægger resultatet i BC
(CD9910 – CALL 4232)		
C9	RET	Returner til BASIC

Programmet startes: PRINT USR 30000.

BC LOAD'es med 13 og puttes op på beregnerstakken. Derefter LOAD'es registeret med 27, som lægges op. 27 ligger nu øverst oppe. Derefter følger RST 28h, som indikerer beregninger. Koden for multiplikation er 04h. Derefter koden 34h, som fortæller, at der ikke er flere beregninger tilbage. Der kan nemlig godt være flere beregninger, men det vender vi hurtigt tilbage til. Efter RST 28h er beregnerstakken ændret. Der er nu kun et tal på stakken, og det er resultatet af beregningen. Det kalder vi ned i BC med CALL A70Eh (1088h), og der returneres. Resultatet 13 x 27, og det er 351. Fik du også det ved afprøvningen?

I næste eksempel vil vi gange 5 med 125 og tage kvadratroden.

Heks-kode	Mnemonic	Forklaring
010500	LD BC,5	Giv BC-registeret værdien 5
CD2015	CALL 5408	Læg det op på beregnerstakken
(CD1817 – CALL 5912)		
017D00	LD BC,125	Værdien 125 lægges i BC
CD2015	CALL 5408	Op med det
(CD1817 – CALL 5912)		
EF	RST 28h	Vi starter beregningerne
04	DEFB	Først multiplikation,
25	DEFB	dernæst kvadratroden
34	DEFB	Vi afslutter beregningerne
CDA70E	CALL 3751	Læg resultatet i BC

(CD8810 – CALL 4232)		
C9	RET	Returner til BASIC

Først lægges 5 og 125 op på stakken. Med RST 28h ganges de først sammen. Der står nu ét tal oppe på beregnerstakken, og det er 625. Derefter tages kvadratroden af det øverste tal på stakken, og det er resultatet af multiplikationen. BC modtager resultatet, og programmet afsluttes. Resultatet udskrives med PRINT USR 30000. Det gav forhåbentlig 25, (SQR (5x125)).

Hvis vi ønsker at trække 40 fra 1000, skal det, der trækkes fra, stakkes sidst, altså:

Heks-kode	Mnemonic	Forklaring
01E803	LD BC,1000	BC får værdien 1000
CD2015	CALL 5408	Op på beregnerstakken med det
(CD1817 – CALL 5912)		
012800	LD BC,40	BC får værdien 40
CD2015	CALL 5408	Op med det
(CD1817 – CALL 5912)		
EF	RST 28h	Vi beregner
03	DEFB	først subtraktion
34	DEFB	Afslutning på beregningerne
CDA70E	CALL 3751	Læg resultatet i BC
(CD8810 – CALL 4232)		
C9	RET	Returner til BASIC

Som du kan se, er koden for subtraktion 03h. Med PRINT USR 30000 får du resultatet: 960.

Hvis vi ønsker 1000:50, gør vi således:

Heks-kode	Mnemonic	Forklaring
01E803	LD BC,1000	BC får værdien 1000
CD2015	CALL 5408	Vi lægger værdien op på beregnerstakken
(CD1817 – CALL 5912)		
013200	LD BC,50	Værdien 50 lægges i BC
CD2015	CALL 5408	Også op med den

	(CD1817 – CALL 5912)	
EF	RST 28h	Beregning:
05	DEFB	Division
34	DEFB	Afslutning på beregning
CDA70E	CALL 3751	Resultatet i BC
	(CD8810 – CALL 4232)	
CD980A	CALL 2712	Udskriv resultatet
	(CD2D0C – CALL 3117)	
C9	RET	Returner til BASIC

Start med RAND USR 30000, da vi for variationens skyld bruger ROM-rutinen, der udskriver indholdet i BC decimalt.

I det næste program vil vi lave følgende udregning:

88 + SQR ((65280 : 255) – 60)

Heks-kode	**Mnemonic**	**Forklaring**
015800	LD BC,88	Giv BC indholdet 88
CD2015	CALL 5408	Læg det op på beregner-stakken
	(CD1817 – CALL 5912)	
0100FF	LD BC,65280	BC får indholdet 65280
CD2015	CALL 5408	Op på beregnerstakken
	(CD1817 – CALL 5912)	
01FF00	LD BC,255	Værdien 255 i BC
CD2015	CALL 5408	Op på beregnerstakken
	(CD1817 – CALL 5912)	
EF	RST 28h	Vi beregner
05	DEFB	Division
34	DEFB	Vi afslutter beregningen
013C00	LD BC,60	Vi giver BC værdien 60
CD2015	CALL 5408	Op på beregnerstakken
	(CD1817 – CALL 5912)	
EF	RST 28h	Beregninger igen
03	DEFB	Først subtraktion
25	DEFB	Kvadratrodsfunktionen
0F	DEFB	Addition
34	DEFB	Vi afslutter

CDA70E	CALL 3751	Resultatet i BC
	(CD8810 – CALL 4232)	
C9	RET	Returner til BASIC

Vi starter med at lægge 88 op på beregnerstakken, da det er det sidste tal, vi skal bruge. Så lægger vi 65280 og 255 derop. Herefter går vi over til beregningerne, hvor vi dividerer 255 op i 65280. Dette resultat står nu øverst på beregnerstakken efterfulgt af 88. Så lægger vi et tredje tal derop, 60, og vi udfører de sidste beregninger: Subtraktionen, kvadratrodsfunktionen og endelig additionen med de 88. Til sidst henter vi resultatet ned i BC-registeret, og det udskrives med PRINT USR 30000, der giver 102. Prøv at regne efter!

Bemærk, at vi ikke lagde 60 derop fra starten. Så ville vi ikke have været i stand til at udføre subtraktionen mellem 256 og 60. Prøv også selv at følge »slagets gang«, og find ud af, hvorfor det ikke kan lade sig gøre.

Koderne for funktionerne, når de bruges efter RST 28h, er:

0F addition
03 subtraktion
04 multiplikation
05 division
25 kvadratrod

Hvis man beregner et tal, der har decimaler, vil det blive rundet af. Det foregår ikke som med INT-funktionen i BASIC, der blot skærer alle decimalerne af, men der rundes naturligt af. Er decimalerne under 0,5 rundes ned – 0,5 eller derover rundes op.

Eksempel:

Heks-kode	**Mnemonic**	**Forklaring**
3E08	LD A,8	Læg værdien 8 i A-registeret
CD1D15	CALL 5405	Læg det op på beregner-stakken
	(CD1517 – CALL 5909)	
EF	RST 28h	Vi starter beregningerne

25	DEFB	Kvadratrodsfunktionen
34	DEFB	Vi afslutter dem
CDA70E	CALL 3751	Vi kalder resultatet ned i BC
(CD8810 – CALL 4232)		
C9	RET	Returner til BASIC

Dette program lægger 8 op på beregnerstakken og tager kvadratroden. Resultatet er 3, da der rundes op. Kvadratroden af 8 er nemlig ca. 2,8!

Bemærk, at vi her gemte 8-tallet i A-registeret og lagde det op med en anden ROM-rutine: 151Dh. Denne ligger tre bytes længere fremme i ROM end 1520h, og disse tre bytes indeholder

```
LD B,0
LD C,A
```

og derefter starter den gamle rutine.

Programmet, der udførte den store udregning, hvor vi behøvede to RST 28h-instruktioner, kan dog gøres nemmere. Hvis der efter RST 28h står koden 01h, vil de 2 øverste tal på beregnerstakken blive ombyttet. Se blot her:

Heks-kode	**Mnemonic**	**Forklaring**
3E58	LD A,88	Læg værdien i A
CD1D15	CALL 5405	Læg det op på beregnerstakken
(CD1517 – CALL 5909)		
3E3C	LD A,60	Giv A indholdet 60
CD1D15	CALL 5405	Læg det op på beregnerstakken
(CD1517 – CALL 5909)		
0100FF	LD BC,65280	Giv BC værdien 65280
CD2015	CALL 5408	Læg det op på beregnerstakken
(CD1817 – CALL 5912)		
3EFF	LD A,255	Giv A værdien 255

CD1D15	CALL 5405	Læg det op på beregnerstakken
(CD1517 – CALL 5909)		
EF	RST 28h	Vi starter beregningerne
05	DEFB	Division
01	DEFB	Ombytning (exchange)
03	DEFB	Subtraktion
25	DEFB	Kvadratrodsfunktionen
0F	DEFB	Addition
34	DEFB	Vi afslutter beregningerne
CDA70E	CALL 3751	Vi lægger resultatet i BC
(CD8810 – CALL 4232)		
C9	RET	Vi returnerer til BASIC

Allerførst lægges nogle tal op på beregnerstakken, der derefter ser således ud:

```
255
65280
60
88
```

Så kommer den første beregning, nemlig divisionen 65280:255, der giver 256. Dette ligger nu øverst på stakken, der ser således ud:

```
256
60
88
```

Så kommer vores 01h, der bytter om.

Nu kan vi subtrahere i den rigtige rækkefølge, nemlig 256-60, og derefter lægges 88 til. Nu ser stakken således ud:

```
102
```

Dette tal lægges ned i BC, og stakken er tom. Du kan forhåbentlig se, hvilken kraft vi her har i 01h-koden. Den kan gøre tingene meget nemmere.

I stedet for at tallene bliver rundet op/ned, så kan vi have brug for decimalerne, specielt med henblik på de trigonometriske funktioner. Det kræver dog en snurrig metode.

I vores SINCLAIR-notation, som også Lambda bruger, er tallene lagret på en helt speciel måde. Prøv bare at PEEK'e programfilen. Så vil du opdage, at hvert tal er opbygget af 5 andre tal, og denne notation er særdeles indviklet. Imidlertid bør vi vide, at også i variabelfilen, hvor ZX 81 holder rede på sine variabelværdier, er de opbygget på denne måde. Dette gælder også beregnerstakken, men det vender vi tilbage til.

Vi vil nu lave et program, der gemmer kvadratroden af 8 i variabelfilen, så vi senere kan kalde værdien frem i BASIC.

Heks-kode	Mnemonic	Forklaring
3E08	LD A,8	Værdien 8 lægges i A
CD1D15	CALL 5405	Læg værdien op på beregnerstakken
(CD1517 – CALL 5909)		
EF	RST 28h	Vi starter beregninger
25	DEFB	Kvadratrod
34	DEFB	Vi afslutter udregningerne
CDF813	CALL 5112	Vi placerer SINCLAIR-notationen
(CDED15 – CALL 5613)		af kvadratroden af 8 i fem registre
2A1040	LD HL,(16400)	Identificerer variabelfilen
23	INC HL	Lægger én til denne adresse
77	LD (HL),A	Kopiér A på denne plads
23	INC HL	Næste adresse
73	LD (HL),E	Kopiér E her
23	INC HL	Næste adresse
72	LD (HL),D	Kopiér D her
23	INC HL	Næste adresse
71	LD (HL),C	Læg værdien af C her
23	INC HL	Næste adresse

70	LD (HL),B	Kopiér B på denne adresse
C9	RET	Returner til BASIC

Først lægger vi værdien 8 op på beregnerstakken. Så tager vi kvadratroden og stopper vore matematiske beregninger. Nu kalder vi en ROM-rutine på adresse 13F8h. Den udfører dette program:

```
LD HL,(16412)
DEC HL
LD B,(HL)
DEC HL
LD C,(HL)
DEC HL
LD D,(HL)
DEC HL
LD E,(HL)
DEC HL
LD A,(HL)
LD (16412),HL
RET
```

Vi finder toppen af beregnerstakken med systemvariablen på adresse 16412 og 16413. Så husker vi, at resultatet af vore beregninger ligger øverst i beregnerstakken, og da det fylder fem bytes (Sinclair-notation), lægger vi hver værdi i de fem bytes ned i hver sit register. Nu skal vi have udskrevet vores tal. Det kan vi desværre ikke gøre i maskinkode – måske kan du finde en metode – men vi kan lægge tallet ned i variabelfilen under et variabelnavn. Vi lægger det ned lige modsat af den måde, vi fandt det på, da vi jo startede fra den forkerte ende. Endelig returnerer vi.

Vi afprøver programmet således: Skriv i BASIC

```
CLEAR
LET X = 0
RAND USR 30000
PRINT X
```

Var svaret kvadratroden af 8? Prøv efter i BASIC.

Behøver vi fortælle, at du selv bør eksperimentere med dette?

*

Vi vil nu se på de andre funktioner, vi kan bruge sammen med beregnerstakken. De andre er, tilført deres RST 28h-koder:

CODE	19	SGN	26
VAL	1A	ABS	27
LEN	1B	PEEK	28
SIN	1C	USR	29
COS	1D	STR$	2A
TAN	1E	CHR$	2B
ASN	1F	NOT	2C
ACS	20	**	06
ATN	21	OR	07
LN	22	AND	08
EXP	23		
INT	24		

CODE, VAL, LEN, STR$ og CHR$ kan umiddelbart være svære at gennemskue. Vi opfordrer dig naturligvis til selv at eksperimentere – bare for rutinens skyld. Et enkelt eksempel vil vi dog give:

Heks-kode	Mnemonic	Forklaring
01FFFF	LD BC,65535	Læg værdien 65535 i BC
CD2015	CALL 5408	Op på beregnerstakken
(CD1817 – CALL 5912)		
EF	RST 28h	Beregninger:
2A	DEFB	Først STR$
1B	DEFB	Dernæst LEN
34	DEFB	Vi afslutter beregninger
CDA70E	CALL 3751	Resultatet ned i BC
(CD8810 – CALL 4232)		
C9	RET	Returner til BASIC

Her får vi udskrevet længden af strengen »65535« svarende i BASIC til PRINT LEN STR$ 65535, og det er naturligvis 5.

AND, OR og NOT – de logiske operatorer – svarer her til BASIC'ens funktioner af samme navn – og de er også svære at bruge. Du må ikke forveksle dem med maskininstruktionerne AND og OR! Eksperimentér selv!

De trigonometriske funktioner kan også bruges. I øvrigt regnes der i radianer. I dette eksempel bruger vi sinus-funktionen. Vi finder SIN 30 og multiplicerer med 20:

Heks-kode	Mnemonic	Forklaring
3E14	LD A,20	A-registeret LOAD'es med 20
CD1D15	CALL 5405	Værdien op på beregnerstakken
(CD1517 – CALL 5909)		
3E1E	LD A,30	A får værdien 30
CD1D15	CALL 5405	Op på beregnerstakken
(CD1517 – CALL 5909)		
EF	RST 28h	Vi beregner:
1C	DEFB	Sinus-funktionen
04	DEFB	Multiplikation
34	DEFB	Vi afslutter beregningerne
CDF813	CALL 5112	Læg SINCLAIR-notationens
(CDED15 – CALL 5613)		fem tal i registre
2A1040	LD HL,(16400)	
23	INC HL	
77	LD (HL),A	
23	INC HL	
73	LD (HL),E	
23	INC HL	Se tidligere program
72	LD (HL),D	
23	INC HL	
71	LD (HL),C	
23	INC HL	
70	LD (HL),B	
C9	RET	Returner til BASIC

Skriv i BASIC:

```
CLEAR
LET X = 0
RAND USR 30000
PRINT X
```

Prøv selv efter i BASIC, om resultatet er korrekt.

Hvis du kender til omvendte trigonometriske funktioner, forstår du, at vi behøver et hjælpeprogram til indtastning af decimaltal.

INT, SGN, ABS, EXP og LN må du selv arbejde med, hvis du vil bruge dem.

Hvis du vil have opløftet et tal til en potens, f.eks. 5^6, så bruges følgende fremgangsmåde:

Heks-kode	Mnemonic	Forklaring
3E05	LD A,5	Først 5 i A
CD1D15	CALL 5405	Op med det
(CD1517 – CALL 5909)		
3E06	LD A,6	Dernæst 6 i A
CD1D15	CALL 5405	Også op med det
(CD1517 – CALL 5909)		
EF	RST 28h	Beregninger:
06	DEFB	Potensopløftning
34	DEFB	Vi afslutter beregninger
CDA70E	CALL 3751	Hent resultatet ned i BC
(CD8810 – CALL 4232)		
C9	RET	Returner til BASIC

PRINT USR klarer sagen. Søger du en forklaring på programmet, må du kigge til højre for det.

Her har vi et eksempel på PEEK-funktionen:

Heks-kode	Mnemonic	Forklaring
3E00	LD A,0	Læg værdien 0 i A-registeret
CD1D15	CALL 5405	Læg den op på beregnerstakken

(CD1517 – CALL 5909)		
EF	RST 28h	Beregninger:
28	DEFB	PEEK-funktionen
34	DEFB	Vi afslutter beregninger
CDA70E	CALL 3751	Resultatet ned i BC
(CD8810 – CALL 4232)		
C9	RET	Returner til BASIC

PRINT USR vil udskrive resultatet, der er 211. Hvorfor? 211 er indholdet på adresse 0 i ROM.

Du kan selv eksperimentere videre med den nuværende viden, men funktionen USR kræver vel en forklaring. Når en sådan mødes, vil der kaldes til den adresse, der står øverst på beregnerstakken. Altså: Står der f.eks. 699 vil USR virke som CALL 699. Efter at CALL er udført, vil de 699 blive erstattet med værdien af BC.

Vi vil her anvise en metode til lagring af tal fra beregnerstakken. I området for systemvariable ligger 30 frie pladser beregnet til formålet. De ligger fra 16477 til 16506 og kaldes MEMBOT. Da vi benytter SINCLAIR-notationen kan vi lagre i alt 6 tal (6*5 = 30). Vi kan gøre det direkte under RST 28h-beregningerne. C0 vil gemme den øverste værdi på beregnerstakken i de første fem bytes, C1 i de næste 5, op til C5, der lagres i de fem sidste.

Vi kan tilbagekalde vores lagrede værdier med E0, E1 op til E5. En vigtig ting ved systemet er, at værdien ikke bliver fjernet i MEMBOT, når den hentes op på beregnerstakken. En lagret værdi kan derfor hentes op på ny, indtil pladsen overtages af en ny værdi. Den værdi, man lagrer, vil *altid* være den, der ligger øverst på beregnerstakken. Også her bliver værdien stående – altså på beregnerstakken! Når der hentes en værdi tilbage, vil den blive lagt øverst på beregnerstakken.

Heks-kode	Mnemonic	Forklaring
3E64	LD A,100	Læg værdien 100
CD1D15	CALL 5405	op på beregnerstakken
(CD1517 – CALL 5909)		
EF	RST 28h	Beregninger:
C5	DEFB	Gem den i 6. position

25	DEFB	Tag kvadratroden
E5	DEFB	Hent de 100 tilbage
04	DEFB	Multiplikation
34	DEFB	Vi afslutter beregningerne
CDA70E	CALL 3751	Resultatet ned i BC
(CD8810 – CALL 4232)		
C9	RET	Returner til BASIC

PRINT USR giver svaret 1000. Hvis du benytter C0 og E0 i stedet for C5 og E5, så går beregningerne ikke. Hvorfor? Vi går ud fra, at det skyldes, at beregningen af kvadratroden selv bruger noget plads i MEMBOT. I hvert fald kan du ikke bruge de 4 første pladser i systemvariablen. Udelader du derimod kvadratrodsberegningen, så kan du sagtens bruge den første position.

Pas på dette under dine beregninger. Arbejd videre selv, for eksempel med et program, der finder COS 1 + SIN 1. Resultatet skal selvfølgelig gemmes i variabelfilen, da det bliver et decimaltal. Brug den fremgangsmåde, vi har anvendt før.

For matematikere vil vi under programeksempler bringe et program, der løser 2.gradsligninger i maskinkode!

*

Det var afslutningen på selve indlæringen af maskinkode. Vi har afsluttet vores kapitler med tips og råd, og nu følger en masse programeksempler. Vi opfordrer dig til selv at udtænke så mange programmer som muligt. Du kan også prøve at effektivisere vore programmer. Vi har med vilje ikke altid valgt den nemmeste og korteste løsning, og vi synes, at du selv bør forbedre vore rutiner. Som nævnt i det forrige kapitel har overskuelighed også en vis betydning, hvilket vi har taget med i vores programforslag. Vi foreslår, at du lagrer vore eksempler i en REM-sætning først i programfilen, så du – som nævnt i sidste kapitel – kan SAVE dem.

Program 1

Vort første program er nok det mest seriøse, vi vil bringe. Det er en skrivemaskine, og programmet ser i al sin enkelhed således ud:

Heks-kode	Mnemonic	Forklaring
FD362200	LD (IY+34),0	Nu kan du bruge alle 24 linier
START		
CDBB02	CALL 699	Keyboardscanning:
(CD3D03 – CALL 829)		
2C	INC L	
20FA	JR NZ,START	Start forfra, hvis tast er aktiveret
VENT		
CDBB02	CALL 699	Keyboardscanning:
(CD3D03 – CALL 829)		
44	LD B,H	
4D	LD C,L	
2C	INC L	
28F8	JR Z,VENT	Start rutinen forfra, hvis ingen tast er aktiveret
CDBD07	CALL 1981	Hvilken er aktiveret?
(CD5E09 – CALL 2398)		
7E	LD A,(HL)	
FE77	CP 119	
C8	RET Z	Returner, hvis RUBOUT (DELETE)
FE76	CP 118	
2804	JR Z,NYLINIE	Gå til ny linie, hvis NEW-LINE (ENTER) er aktiveret
CB77	BIT 6,A	

20E3	JR NZ,START	Hop til start, hvis tegnet ikke er displaybart
NYLINIE		
D7	RST 10h	Udskriv tegn eller skift linie
18E0	JR START	Begynd forfra

Prøv selv at lave programmet med en markør, og få det til at udskrive grafiske tegn. Du kan også bruge EDIT til at slette tegn med. For at gøre det rigtigt professionelt, så lav en rutine, der bevirker, at man kan trække linier sammen. Der er muligheder nok, og udnytter du dem alle, får du starten på et tekstbehandlingsprogram.

Program 2

Vi vil her lave et program, (gemt fra adresse 16514 (17307) som sædvanlig) der finder rødderne i en 2. gradsligning af formen

$$ax^2 + bx + c = 0$$

Hvis ligningen hedder

$$x^2 + 2x - 8 = 0$$

er a = 1, b = 2 og c = −8.

For at finde rødderne, (der er 2 af slagsen) skal vi bruge den matematiske formel:

$$x_1 = \frac{-b + SQR\,(b^2-4ac)}{2a}$$

$$\text{og } x_2 = \frac{-b - SQR\,(b^2 - 4ac)}{2a}$$

Programmet skal altså inden udregningen have at vide, hvilke værdier a, b og c antager. Da de godt kan være negative eller decimaltal, eller større end 65535, kan vi ikke LOAD'e BC med tallet og kalde adresse 1520h i ROM. Så en anden udvej er at gøre følgende i BASIC:

```
CLEAR
LET A = a
LET B = b
LET C = c
```

Disse tre tal er nu gemt i variabelfilen, hvorfra de skal overføres til MEMBOT-området, hvor de skal ligge på pladserne 3, 4 og 5

for hhv. a, b og c. Med CLEAR-kommandoen slettedes hele variabelfilen, så nu indeholder den kun de tre indtastede tal. Den første adresse i variabelfilen indeholder nu en værdi, der fortæller, hvilket navn den første variabel har. De næste fem bytes fortæller, hvilken værdi variablen har. Dette er skrevet i SINCLAIR-NOTATION. Disse første seks bytes indeholder i dette tilfælde altså en byte fortællende, at navnet på den første variabel er A, samt værdien a. Efter disse seks bytes følger endnu 2*6 bytes – nemlig 6 for variablen B og 6 for C.

De fem bytes, der viser hvilken værdi, A har, skal overføres til Membot plads 3. Desuden overføres de 5 bytes for B og de 5 for C. Ialt overføres altså 3*5 bytes til MEMBOT-pladserne 3, 4 og 5. MEMBOT har følgende opbygning:

adresse 16477-16481 – plads 0
adresse 16482-16486 – plads 1
adresse 16487-16491 – plads 2
adresse 16492-16496 – plads 3
adresse 16497-16501 – plads 4
adresse 16502-16506 – plads 5

Vi skal altså overføre A, B og C til 16492-16506 fra variabelfilen. Ved overførslen til MEMBOT skal vi bruge dette program:

Heks-kode	**Mnemonic**	**Forklaring**
116C40	LD DE,16492	Første adresse i MEMBOT plads 3
2A1040	LD HL,(16400)	Første adresse i variabelfil
E5	PUSH HL	Gem HL på stakken
0603	LD B,3	Antal variable
LØKKE		
C5	PUSH BC	Gem BC på stakken
23	INC HL	Spring byten, som indikerer variablens navn, over
010500	LD BC,5	Et tal fylder fem bytes
EDB0	LDIR	Overfør data
C1	POP BC	Hent BC tilbage
10F6	DJNZ LØKKE	Er der flere variable?

Nu kan vi gå i gang med udregningerne. Inden vi gør det, så læg mærke til, at størrelserne 2a og kvadratroden af b^2-4ac går igen i begge løsninger. Vi vil under gennemgangen konstant i højre side vise, hvordan beregnerstakken ser ud. Allerførst lægger vi 2 op på den.

3E02	LD A,2	
CD1D15	CALL 5405	Stak: 2
(CD1517 – CALL 5909)		

Så kommer vi til RST 28h, og først af alle beregninger henter vi c op fra MEMBOT (c står i position 5).

EF	RST 28h	
E5	DEFB	Stak: c
		2

Så byttes de to øverste værdier om.

01	DEFB	Stak: 2
		c

2 lægges ned i position 2 og hentes op igen.

C2	DEFB	
E2	DEFB	Stak: 2
		2
		c

Så hentes a op fra position 3 i MEMBOT.

E3	DEFB	Stak: a
		2
		2
		c

Så multiplicerer vi a og 2, og lægger resultatet ned i MEMBOT plads 5.

04	DEFB	
C5	DEFB	Stak: 2a
		2
		c

Så multiplicerer vi.

04	DEFB	Stak: 4a
		c

Endnu en gang.

04	DEFB	Stak: 4ac

Nu henter vi b op 2 gange (fra position 4 i MEMBOT).

E4	DEFB	
E4	DEFB	Stak: b
		b
		4ac

b^2 er lig b*b, så derfor ganges

04	DEFB	Stak: b^2
		4ac

Nu skal vi udføre subtraktionen $b^2 - 4ac$, men vi skal huske at bytte dem om for at få subtraheret rigtigt.

01	DEFB	
03	DEFB	Stak: $b^2 - 4ac$

Så tager vi kvadratroden, og gemmer den i plads 3.

25	DEFB	
C3	DEFB	Stak: $\sqrt{b^2 - 4ac}$

Denne størrelse skulle vi bruge i udregningen. Nu skal vi gøre denne udregning

$$-b + \sqrt{b^2 - 4ac}$$

men det er jo det samme som

$$\sqrt{b^2 - 4ac} - b$$

Derfor hentes b op, og vi subtraherer.

E4	DEFB	
03	DEFB	Stak: $-b+\sqrt{b^2-4ac}$

Nu mangler vi bare at dividere med 2a, og netop dette tal har vi på plads 5.

E5	DEFB	Stak: $\frac{(-b+\sqrt{b^2-4ac})}{2a}$
05	DEFB	
34	DEFB	

Som du ser, er den ene løsning nu på toppen af beregnerstakken. Denne løsning skal vi have ned i variabelfilen, og vi kan passende gemme den under den første variabel i filen, A. I starten af programmet gemte vi HL, som indeholdt startadressen for variabelfilen.

Heks-kode	Mnemonic	Forklaring
CDF813	CALL 5112	Overfør resultatet til registrene A, B, C, D og E
(CDED15 – CALL 5613)		
E1	POP HL	Hent HL ned

Så kalder vi den sædvanlige rutine, som kaldes OVERFØR, der overfører resultatet, som nu står i registrene A, B, C, D og E, til variabelfilen. Rutinen er gennemgået i kapitlet om beregnerstakken og vil blive vist igen senere.

```
CDCA40          CALL OVERFØR Kald overførselsrutine
  (CDE343 – CALL OVERFØR)
E5              PUSH HL      Gem HL
```

HL peger nu til den sidste byte af BASIC-variabel A. Nu skal næste løsning udregnes.

```
3E00            LD A,0
CD1D15          CALL 5405    Stak: 0
  (CD1517 – CALL 5909)
EF              RST 28h
```

Bemærk, at vor løsning nummer ét stadig står på stakken. Vi kan dog ikke udnytte den. I stedet henter vi $\sqrt{b^2 - 4ac}$, som er gemt i plads 3.

```
E3              DEFB         Stak: √(b²–4ac)
                                   0
```

Hvis vi nu foretager en subtraktion, vil resultatet blive

$0-\sqrt{b^2-4ac}$, og det er $-\sqrt{b^2-4ac}$

```
03              DEFB         Stak: –√(b²–4ac)
```

Så henter vi b op og subtraherer.

```
E4              DEFB
03              DEFB         Stak: –b–√(b²–4ac)
```

Til sidst mangler vi at dividere med 2a.

```
E5              DEFB
05              DEFB         Stak: (–b–√(b²–4ac))
                                        2a
34              DEFB
```


Anden løsning står nu på toppen og skal overføres til variablen B i variabelfilen.

```
CDF813          CALL 5112    Overfør resultatet til
  (CDED15 – CALL 5613)       registrene A, B, C, D og E
E1              POP HL       Hent HL, der peger på sid-
                             ste byte under variablen A
```

HL skal springe to bytes over for at nå næste variabel. De to, der skal springes over, er sidste byte i A, og byten, som indikerer navnet på næste variabel, dvs. B.

```
23              INC HL       Flyt HL til den
OVERFØR
23              INC HL       rigtige position
```

Læg mærke til, at rutinen OVERFØR først starter ved den anden INC HL. Nu følger vores gamle prodedure:

```
77              LD (HL),A
23              INC HL
73              LD (HL),E
23              INC HL
72              LD (HL),D
23              INC HL
71              LD (HL),C
23              INC HL
70              LD (HL),B
C9              RET
```

Efter at programmet er indtastet, prøv at følge vores eksempel:

Løs ligningen $x^2+2x-8 = 0$

Altså:

```
10 CLEAR
20 LET A=1
```

```
30 LET B=2
40 LET C=-8
50 RAND USR 16514 (17307)
PRINT A,B
```

Du kan evt. erstatte linie 20-40 med sætningerne

```
20 INPUT A
30 INPUT B
40 INPUT C
```

Løsningerne kan du selv regne efter ved brug af de viste formler.

Prøv i øvrigt at bemærke, hvor hurtigt ZX 81 regner med tallet nul. Lad A være 1, B nul og C nul, og kør programmet. Resultatet bliver nul, der er dobbeltrod.

Program 3

Dette program er ligesom det foregående et matematisk program. Det tegner en sinuskurve, men kan naturligvis også tegne andet, hvis du justerer det. Vi har placeret det fra adresse 16514 (17307). Programmet er meget enkelt, og forklaringerne til programmet er kun angivet til højre for programmets mnemonics.

Heks-kode	Mnemonic	Forklaring
01002B	LD BC,11008	B er 43, C er 0
C5	PUSH BC	
CDB60B	CALL 2998	Plot koordinatsystemets y-akse
(CD3D0D – CALL 3389)		
C1	POP BC	
10F9	DJNZ –7	
01000A	LD BC,2560	B er 10, C er 0
CDF508	CALL 2293	
(CD940A – CALL 2708)		
3E82	LD A,130	
D7	RST 10h	Tegn x-aksen
3C	INC A	
061F	LD B,31	
D7	RST 10h	
10FD	DJNZ –3	
2A1040	LD HL,(16400)	
23	INC HL	
117640	LD DE,16502	Overfør pi til MEMBOT, plads 6
010500	LD BC,5	
EDB0	LDIR	
3E16 LØKKE	LD A,22	
CD1D15	CALL 5405	

```
  (CD1517 – CALL 5909)
3E14          LD A,20
CD1D15        CALL 5405      Læg tal op på beregnerstak-
                             ken
  (CD1517 – CALL 5909)
3E00          LD A,0
CD1D15        CALL 5405
  (CD1517 – CALL 5909)
3E20          LD A,32
CD1D15        CALL 5405
  (CD1517 – CALL 5909)
EF            RST 28h
05            DEFB DIV
E5            DEFB HENT I 5
04            DEFB MULT      Beregninger
1C            DEFB SIN
04            DEFB MULT
0F            DEFB ADD
34            DEFB END
CDA70E        CALL 3751
  (CD8810 – CALL 4232)
41            LD B,C
21B240        LD HL,16562
  (21CB43 – LD HL,17355)
4E            LD C,(HL)      Plot punkt
34            INC (HL)
CDB60B        CALL 2998
  (CD3D0D – Call 3389)
C3A740        JP LØKKE       Hop tilbage
  (C3C043 – JP LØKKE)
```

Programmet er en forbedring af et lignende nævnt i den danske instruktionsbog til ZX 81. Gå selv det hele igennem, sæt dig grundigt ind i tingene, og forklar det for dig selv.

Programmet køres således:


```
POKE 16562,0 (dog kun ved genkørsel)
  (POKE 17355,0)
CLEAR
LET L=PI
RAND USR 16514
  (RAND USR 17307)
```

Programmet indeholder ingen tæller, der holder øje med, hvor mange gange der PLOT'tes. Derfor vil programmet slutte med fejlmeddelelsen b/0 (IR), når vi forsøger at PLOT'te uden for skærmbilledet.

Byte 16562 (17355) bruger programmet under sine beregninger, og den skal nulstilles efter hver kørsel.

Prøv dette:

```
POKE 16575,29
  (POKE 17368,29)
```

Hvad skete der? Find selv adresse 16575 (17368), og forklar det skete.

Du kan også prøve at finde på andre matematiske grafer, og derved blot ændre vore beregninger.

Program 4

Det næste i rækken er et spil, som hører hjemme i kategorien familiespil. Første gang du ser dette spil, synes du sikkert ikke, at det ser sjovt ud, men prøv det selv: og snart vil du være i konkurrence med hele familien om at få rumfartøjerne sikkert ned til landing.

Vi har kaldt det MOON LANDER, selv om det ikke umiddelbart fremgår af programmet. Du kan selv skabe en forside eller indflette dit eget navn i skærmens linier.

Vi har valgt denne gang at bringe programmet i sin helhed, men det forhindrer os naturligvis ikke i at kommentere det. Vi vil kigge på de forskellige rutiner og få dem til at fungere. Følg selv med i programmet lidt længere fremme.

Først skal du vide, hvad der er ideen bag dette. Der kommer et rumskib, et A, flyvende tværs over skærmen. På et tidspunkt kan du – ved at trykke på BREAK-tasten – få dit rumskib til at gå lodret ned. Her »lander« det, men det kommer kun sikkert ned, hvis det ikke »lander« i et andet rumskib. Da der er plads til 32 rumskibe, kan du maksimalt få 32 points. Du kan selv – efter din fantasis bedste evner – indflette bonuspoints og meget andet: du kan også få skærmen til at skifte karakter. Kun din fantasi sætter grænser. Lander du i et rumskib stopper spillet. Her kan du indsætte kommentarer om indsatsen etc..

Nu kigger vi på programmet. Det fylder 245 bytes. Det skal du indtaste i en REM-sætning *først* i programmet. De første tegn giver du påskriften POINTS: – du skal bruge denne skrift i programmet. Herefter indtaster du heks-koderne i rigtig rækkefølge. På Lambda kan du ikke skrive POINTS: direkte, da den opfatter INT som BASIC-funktionen af samme navn. Kolon findes heller ikke. Det første problem kan du klare ved at POKE'e bogstaverne ind, og det andet kan du løse ved at erstatte kolon med lighedstegn.

Den første rutine ligger fra adresse 16521 (17314) – her kalder du med RAND USR – og den fylder kun ni bytes. Den giver udskriften POINTS: øverst på skærmen. Vi benytter ROM-rutinen på 2923 (3318), der udskriver en streng.

Det næste skridt er at få udskrevet to streger på skærmen. Vi skal have udskrevet én lige under vore points, og endelig én til at lande på. Ved hjælp af en løkke får vi udskrevet 32 tegn på 2. linie af typen med koden 128 – en sort firkant. Så hopper vi ned til 21. linie. Rumskibene skal lande på linien ovenover. Med en løkke udskrives 32 tegn af typen med koden 3.

En rutine på 13 bytes er den næste. Bemærk dog først, at vi ved stregtegningen har lagt udskriftfilens startadresse op på stakken. Nu fastlægger vi vor printposition med ROM-rutinen 2293 (2708) – sammenlign selv med beskrivelsen i kapitlet om disse rutiner – og her udskriver vi vor pointscore, der fra start naturligvis er nul. Udskriften klares med decimalrutinen på 2712 (3117). Bemærk også her, at vi har lagt pointtallet op på stakken. Nu ligger der to værdier deroppe, men kun pointværdien er umiddelbar tilgængelig.

I byte 16569 (17362) starter selve programmet. De tre forrige rutiner har kun været med for at initialisere skærmen. Her finder vi et tilfældigt tal i systemvariabel 16436, og vi sikrer os, at det bliver mellem 3 og 18. Vi multiplicerer det med 33 (antallet af tegn pr. linie) og lægger det til den fra stakken nedhentede variabelfil-værdi. Bemærk, at antallet af points midlertidigt lægges i DE, for umiddelbart at blive lagt øverst igen. Ved udgangen af rutinen peger HL på det newline, der er før den linie, der er tilfældigt udvalgt.

Nu skal vi hen ad skærmen. Først incrementerer – vi skal jo ikke POKE'e newline. Så finder vi ud af, om vi nu peger på en newline, for da vil vi være ved enden af linien, og vi *skal* ned ad skærmen. Der hoppes derfor til rutinen NED. Hvis det ikke er newline, POKE'er vi et A, holder en pause, og sletter A igen. Nu kalder vi BREAK-rutinen. Er BREAK ikke nedtrykket, vil C-flaget være sat, og der returneres. Er den aktiveret, går vi til rutinen NED II.

Her ser vi forskellen på rutinen NED og NED II. I NED-rutinen tilføjer vi et DEC HL. Vi husker, at vi kun kaldte denne rutine,

hvis vi var på et newline, og vi skal derfor decrementere, så vi kommer på skærmens sidste plads på den tilfældigt valgte linie. Så fortsætter vi med NED II, som vi havde brugt, hvis vi selv havde aktiveret »nedturen«.

Selve NED II-rutinen er en lidt klodset sag, og måske kan du lave det smartere. Vi lægger 33 til positionen for at komme en linie længere ned. Hvis der står et tegn med koden 3, så er vi landet (vi står på landingspladsen). Er vi ikke landet, sammenligner vi med et andet tegn, nemlig endnu et A. Møder vi et A på nedturen – inden vi lander – så er vores manøvre mislykket, og vi kalder rutinen RAMT. Hvis vi ikke er landet eller har ramt et andet rumskib, skal vi skrive A, holde pause, slette det igen, for endelig at hoppe tilbage til det sted i programmet, hvor vi går en linie længere ned.

Bemærkede du, at hvis rumskibet var landet, var det ikke skrevet på skærmen alligevel! I KOLONNE A-rutinen slettede vi det jo, og det blev aldrig skrevet i NED II-rutinen uden at blive slettet, hvis det endelig var skrevet. Det råder vi bod på nu.

Først erindrer vi, at vi faktisk peger på linien med »landingslinien«, så først må vi en linie længere tilbage. AND A sletter blot C-flaget. Nu kan vi skrive rumskibet ind på landingspladsen. Så gør vi klar til pointudskrift. Vi finder den rigtige printposition, henter pointværdien ned fra stakken, incrementerer den og lægger den op, for sluttelig at udskrive den. Nu holder vi en pause beroende på et tilfældigt tal mellem 0 og 65535. Måske synes du, at en pause på f.eks. ét gennemløb er for lidt, men så kan du jo indlægge en OR-instruktion, der sikrer en minimumspause.

Efter dette henter vi vor kære udskriftfils begyndelsesadresse ned igen, og vi adderer, til vi når vor landingsplads. Her undersøger vi hele linien med en nyttig CPIR-ordre. Er der nogle mellemrum? Er der det, så er Z-flaget sat, og vi returnerer til endnu et rumskib. Er der ikke nogen mellemrum, så har vi fået landet 32 rumskibe – maksimumantallet – og vi hopper til GAME OVER.

Den næste rutine er den, vi kalder, hvis vi rammer et andet rumskib. Her sker der nogle mærkelige ting og sager: faktisk skrives blot nogle stjerner, der udgør vores »crash«!

Efter denne fyldige rutine er spillet forbi, og vi møder GAME

OVER-rutinen, som vi også kaldte, hvis vi havde maksimumpoint. Her skal vi blot returnere til BASIC, men vi husker vel, at vi har to værdier liggende på vores stak, som skal fjernes, før vi kan komme ned til returadressen. Vi POP'er de to værdier og returnerer.

Vi har nu gennemgået programmet så grundigt som muligt uden at gå i detaljer. Du har forhåbentlig fået masser af gode ideer til ændringer og udvidelser – brug dem! Selv om de måske ikke går godt de første ti gange, så prøv 11. gang med!

Som tips til spillet kan vi da kort nævne, at de to rumskibe i »enderne« kan du hurtigt få. Som beskrevet kan du få den længst til højre på skærmen ved blot at lade rumskibet »køre«. Det første får du ved at holde BREAK nede, inden rumskibet kommer på skærmen. Men pas på: Nogle gange er pausen imellem rumskibenes ankomst så lille, at hvis du holder BREAK for lang tid nede, når du aktiverer, får du også aktiveret næste rumskib, der så lander på den første plads. Og er denne allerede fyldt ud, så »sidder du i saksen«. Derfor er det nok en god idé at indlægge den omtalte minimumspause, men det er op til dig selv: tippet har du fået under gennemgangen. I 16604 og 16605 (17397 og 17398) ligger pausestørrelsen mellem hvert A, der går henover skærmen. Prøv at ændre hastigheden. Hvis du mindsker værdien, kører rumskibet hurtigere, og vice versa.

Endelig: får du over 20 points de første timer (eller dage), du prøver, så er dit øjemål ikke så slet endda. Sidder du lang tid foran skærmen, bliver du naturligvis hurtig god. Når du de omtalte 32 points, så lad os det vide!

Heks-kode		Mnemonics
118240	OVERSKRIFT	LD DE,16514
(119B43 – LD DE,17307)		
010700		LD BC,7
CD6B0B		CALL 2923
(CDF60C – CALL 3318)		
2A0C40	STREGER	LD HL,(16396)
(217D40 – LD HL,16509)		
E5		PUSH HL

```
012100                  LD BC,33
09                      ADD HL,BC
0620                    LD B,32
23                      INC HL
3680                    LD (HL),128
10FB                    DJNZ -5
015302                  LD BC,595
09                      ADD HL,BC
0620                    LD B,32
23                      INC HL
3603                    LD (HL),3
10FB                    DJNZ -5
010800          POINTS  LD BC,8
CDF508                  CALL 2293
  (CD940A - CALL 2708)
010000                  LD BC,0
C5                      PUSH BC
CD980A                  CALL 2712
  (CD2D0C - CALL 3117)
3A3440          LINIE A LD A,(16436)
E60F                    AND 15
C603                    ADD A,3
CD1D15                  CALL 5405
  (CD1517 - CALL 5909)
3E21                    LD A,33
CD1D15                  CALL 5405
  (CD1517 - CALL 5909)
EF                      RST 28h
04                      DEFB MULT
34                      DEFB END
CDA70E                  CALL 3751
  (CD8810 - CALL 4232)
D1                      POP DE
E1                      POP HL
E5                      PUSH HL
D5                      PUSH DE
09                      ADD HL,BC
23          KOLONNE A   INC HL
```



```
3E76                    LD A,118
BE                      CP (HL)
2813                    JR Z, NED
3626                    LD (HL),38
110005                  LD DE,1280
1B                      DEC DE
7B                      LD A,E
B2                      OR D
20FB                    JR NZ,-5
3600                    LD (HL),0
CD460F                  CALL 3910
  (CD3B11 - CALL 4411)
38E9                    JR C, KOLONNE A
1801                    JR NED II
2B              NED     DEC HL
012100          NED II  LD BC,33
09                      ADD HL,BC
3E03                    LD A,3
BE                      CP (HL)
2813                    JR Z, LANDET
3E26                    LD A,38
BE                      CP (HL)
283F                    JR Z, RAMT
3626                    LD (HL),38
110005                  LD DE,1280
1B                      DEC DE
7B                      LD A,E
B2                      OR D
20FB                    JR NZ,-5
3600                    LD (HL),0
18E4                    JR NED II
A7              LANDET  AND A
ED42                    SBC HL,BC
3626                    LD (HL),38
010800                  LD BC,8
CDF508                  CALL 2293
  (CD940A - CALL 2708)
```

```
C1                          POP BC
03                          INC BC
C5                          PUSH BC
CD980A                      CALL 2712
  (CD2D0C – CALL 3117)
3A3440                      LD A,(16436)
57                          LD D,A
3A3440                      LD A,(16436)
5F                          LD E,A
1B                          DEC DE
7B                          LD A,E
B2                          OR D
20FB                        JR NZ,–5
C1                          POP BC
E1                          POP HL
E5                          PUSH HL
C5                          PUSH BC
017402                      LD BC,628
09                          ADD HL,BC
012000                      LD BC,32
3E00                        LD A,0
EDB1                        CPIR
2880                        JR Z, LINIE A
1839                        JR GAME OVER
3617            RAMT        LD (HL),23
ED42                        SBC HL,BC
23                          INC HL
3E76                        LD A,118
BE                          CP (HL)
2802                        JR Z,2
3617                        LD (HL),23
2B                          DEC HL
2B                          DEC HL
BE                          CP (HL)
2802                        JR Z,2
3617                        LD (HL),23
012000                      LD BC,32
ED42                        SBC HL,BC
```



```
3617                        LD (HL),23
110028                      LD DE,10240
1B                          DEC DE
7B                          LD A,E
B2                          OR D
20FB                        JR NZ,–5
014100                      LD BC,65
ED42                        SBC HL,BC
3E76                        LD A,118
BE                          CP (HL)
2802                        JR Z,2
3617                        LD (HL),23
2B                          DEC HL
2B                          DEC HL
BE                          CP (HL)
2802                        JR Z,2
3617                        LD (HL),23
ED42                        SBC HL,BC
3617                        LD (HL),23
C1              GAME OVER   POP BC
E1                          POP HL
C9                          RET
```

Program 5

Vi vil nu lave det omtalte SPACE INVADERS-spil. Hvis du ikke kender spillet fra spillemaskiner, vil vi kort fortælle dig om det. Du har en base nederst på skærmen, som du skal forsvare mod nogle rumskibe. Disse rumskibe starter fra toppen af skærmen og går længere og længere ned mod dig. Det er din opgave at skyde dem ned, inden de når din base. Rumskibene skyder også på dig, og du kan blive skudt et forudbestemt antal gange.

Programmet fylder ca. 1300 bytes (over 1 K), og vi vil derfor ikke vise en samlet fremstilling af programmet. Det vil blive gennemgået trin for trin, og heks-koderne bliver vist i brudstykker. Vi anbefaler, at du indtaster heks-koderne, efterhånden som de dukker op. Det vil ikke være sjovt at skulle indtaste 1300 heks-koder (det er 2600 anslag) i en omgang. Programmet vil blive lagret i den sædvanlige REM-sætning, så du kan gemme det på bånd.

Det første, du gør, er at indtaste følgende linie:

```
1 REM XXXX (94 tegn)
```

Derefter laver du 12 linier til af samme slags – brug EDIT. Nu har du så linierne 1-13 med ovenstående indhold, og du skriver som direkte kommandoer:

```
POKE 16511,16          (POKE 17304,16)
POKE 16512,5           (POKE 17305,5)
POKE 16514,118         (POKE 17307,118)
POKE 16515,118         (POKE 17308,255)
POKE 16510,0           (POKE 17303,0)
```

Du har nu en linie 0 med 5 x 256 + 16 = 1296 tegn. Så indtaster du heks-loaderen. Linie 10 skal ændres til


```
10 LET X = 16516   (LET X = 17309)
```

Vi har jo brugt 16514-16515 (17307-17308) til at forhindre crash ved automatiske listninger. Programmet følger nu, og *alle* viste heks-koder skal indtastes. Hvis du udelader nogle, vil CALL- og JP-adresser jo ikke passe.

SPACE INVADERS

Den første byte er DATA, som programmet skal bruge. Programmet skal vide, hvordan invaderne (som rumskibene vil blive kaldt) ser ud. Vi vil bruge CHR$ 6; CHR$ 134. I en række ved starten er der 8 af slagsen, og der skal selvfølgelig være mellemrum mellem dem.

```
000686000686 DATA      DEFB INVADERS
000686000686           DEFB INVADERS
000686000686           DEFB INVADERS
00068600068618         DEFB INVADERS
```

Vi har også brug for at skrive »SCORE=«, »LIV=« og »*SLUT*«. Koderne for S, C, O, osv. skriver vi her:

```
382834372A14           DEFB SKRIFT
18312E3B1418           DEFB SKRIFT
979780B8B1BA           DEFB SKRIFT
B980979718             DEFB SKRIFT
```

Du har måske opdaget, at ordene er skilt ad med et divisionstegn (kode 24 – 18h). Hvorfor finder du ud af senere.

Dit rumskib vil være CHR$ 0; CHR$ 135; CHR$ 128; CHR$ 4; CHR$ 0.

```
008780040018           DEFB RUMSKIB
```

Bemærk, at det er omgivet af mellemrum. Til sidst skal der reserveres plads til antal points, liv osv., i alt 7 bytes. Det er lige meget, hvad der står, de defineres senere.

```
10203040506070          DEFB DATA
```

Nu til selve programmet. Der vil blive lavet underrutiner, som bevæger dig, bevæger invadernes bomber, invaderne selv osv. Disse rutiner vil blive kaldt fra en hovedrutine, som vi kalder DRIVER.

Først har vi brug for en lille rutine, der hedder PRINT (HL).

```
7E          PRINT (HL)  LD A,(HL)
23                      INC HL
FE18                    CP 24
C8                      RET Z
D7                      RST 10h
18F8                    JR -8
```

Denne rutine udskriver en streng, og HL er, før rutinen kaldes, LOAD'et med strengens startadresse. Som du ser, bliver A-registeret LOAD'et med (HL). Hvis A er 24, hvilket er divisionstegnet, returneres. Derfor blev vores DATA adskilt med et sådant tegn. Hvis A ikke er 24, bliver A værdien udskrevet, og der startes om igen via JR -8 med næste (HL)-værdi.

Næste rutine laver skærmen, dvs. tegner ivader'ne, basen, »SCORE« osv.

```
010100          SKÆRM   LD BC,1
CDF508                  CALL 2293
  (CD940A - CALL 2708)
219D40                  LD HL,16541
  (21B643 - LD HL,17334)
CDC140                  CALL PRINT (HL)
  (CDDA43 - CALL PRINT (HL))
```

Du husker forhåbentlig, hvordan CALL 2293 (2708) virkede.

I programmet ændres printpositionen til 0,1. På adresse 16541 (17334) starter »SCORE=«, og dermed skrives AT 0,1; »SCORE=«. Vi skal også skrive »LIV=« AT 0,25. »LIV=« starter på 16548 (17341), så vi skriver således:


```
011900                  LD BC,25
CDF508                  CALL 2293
  (CD940A - CALL 2708)
21A440                  LD HL,16548
  (21BD43 - LD HL,17341)
CDC140                  CALL PRINT (HL)
  (CDDA43 - CALL PRINT (HL))
```

Så skal vi lave en streg på linie 1 og linie 23. Vi bruger dermed hele skærmen. Vi laver senere en rutine kaldet CHR3, som skriver tegnet med koden 3 så mange gange, som register B angiver. I en linie er der 32 kolonner.

```
010001                  LD BC,256
CDF508                  CALL 2293
  (CD940A - CALL 2708)
0620                    LD B,32
CD2F41                  CALL CHR3
  (CD4844 - CALL CHR3)
010017                  LD BC,5888
CDF508                  CALL 2293
  (CD940A - CALL 2708)
0620                    LD B,32
CD2F41                  CALL CHR3
  (CD4844 - CALL CHR3)
```

I linie 20 skal vi lave 4 skjolde til dig, hvert bestående af 3 CHR3-tegn, og der skal være 4 mellemrum mellem dem.

```
010414                  LD BC,5124
C5          L1          PUSH BC
CDF508                  CALL 2293
  (CD940A - CALL 2708)
0603                    LD B,3
CD2F41                  CALL CHR3
  (CD4844 - CALL CHR3)
3E07                    LD A,7
C1                      POP BC
```

```
81                      ADD C
4F                      LD C,A
FE20                    CP 32
20EE                    JR NZ, L1
```

Nu skal vi have lavet invader'ne. De skal være i starten af linie 3, 5, 7, 9 og 11. Du husker, at de begynder på adresse 16516 (17309).

```
010103                  LD BC,769
C5          L2          PUSH BC
CDF508                  CALL 2293
  (CD940A – CALL 2708)
218440                  LD HL,16516
  (219D43 – LD HL,17309)
CDC140                  CALL PRINT (HL)
  (CDDA43 – CALL PRINT (HL))
C1                      POP BC
04                      INC B
04                      INC B
3E0D                    LD A,13
B8                      CP B
20EE                    JR NZ, L2
```

BC LOAD'es med 769 (AT 3,1) og PUSH'es. Så kaldes 2293 (2708), og printpositionen ændres til AT 3,1. Så tegnes de 8 invader'e og BC POP'es tilbage. B-registeret, der passer på linienummeret, forøges med 2. Hvis B ikke er 13, hoppes tilbage til PUSH BC, og rutinen køres igen. Hvis B er blevet 13, skulle der jo ikke være flere invadere, og der fortsættes i rutinen. Næste og sidste punkt er at lave din kanon. Da vi reserverede 7 bytes i DATA kan vi gemme dens position her, og det bliver på adresserne 16572 (17365) for linienummer og 16573 (17366) for kolonnenummer. Der står intet af værdi på disse bytes endnu, men det kommer senere.

```
ED4BBC40                LD BC,(16572)
  (ED4BD543 – LD BC,(17365))
```



```
CDF508                  CALL 2293
  (CD940A – CALL 2708)
21B440                  LD HL,16564
  (21CD43 – LD HL,17357)
CDC140                  CALL PRINT (HL)
  (CDDA43 – CALL PRINT (HL))
C9                      RET
```

Printpositionen ændres til kanonens position. På adresse 16564 (17357) står kanonens udseende. Til sidst returneres til CALL-stedet. Nu til CHR3-rutinen, og den behøver forhåbentlig ingen forklaring.

```
3E03            CHR3    LD A,3
D7                      RST 10h
10FD                    DJNZ –3
C9                      RET
```

Den næste rutine er KEYBSCAN, der finder ud af, om du bevæger din kanon til højre eller venstre, eller om du skyder. Vi vil bruge tasterne 2 og 4 til hhv. venstre og højre. 1, 3 og 5 vil dog også få din kanon til at køre til højre. Hvis én af disse trykkes ned, vil bit 3 i register L blive resat. Hvis du ikke forstår hvorfor, må du kigge i kapitlet om keyboardscanningen. Til at affyre kanonen bruges Ø-tasten, bit 4 i L resættes. Hvis 2 nedtrykkes, er bit 2 i H resat, og hvis 4 bit 4 i H.

KEYBSCAN

```
CDBB02          CALL 699        Kald keyboardscanningen
  (CD3D03 – CALL 829)
E5              PUSH HL         Gem HL på stakken
CB5D            BIT 3,L         Er 1, 2, 3, 4 eller 5 nedtrykket?
2023            JR NZ,SKUD      Hvis ikke, hop til SKUD
ED4BBC40        LD BC,(16572)   LOAD B med linienr., C
                                med kolonnenr.
  (ED4BD543 – LD BC,(17365))
CB54            BIT 2,H         Er 2 nedtrykket?
```

```
2008            JR NZ,HØJRE     Hop til HØJRE, hvis nej
0D              DEC C           decrementer kolonnenr. (til
                                venstre)
79              LD A,C          Hvis C var 0, kan kanonen
                                ikke komme længere til
FEFF            CP 255          venstre, der lades som in-
2815            JR Z,SKUD       genting, og der hoppes til
                                SKUD
1806            JR NYPOS        Den kan komme til venstre,
                                så hop
0C      HØJRE   INC C           Kan kanonen komme læn-
                                gere til højre?
79              LD A,C          Kanonen fylder fem tegn.

FE1C            CP 28           og derfor sammenlignes
                                med 28
280D            JR Z,SKUD       Hvis ikke, hop til SKUD
        NYPOS
ED43BC40        LD (16572),BC   Gem den nye position
  (ED43D543 – LD (17365),BC)
CDF508          CALL 2293       Sæt kanonposition lig print-
                                position
  (CD940A – CALL 2708)
21B440          LD HL,16564     Startadresse på kanonens
                                udseende
  (21CD43 – LD HL,17357)
CDC140          CALL PRINT (HL) Tegn kanonen
  (CDDA43 – CALL PRINT (HL)
2A0C40 SKUD     LD HL,(16396)   Undersøg skærmen for, om
                                der er
  (217D40 – LD HL,16509)
01D602          LD BC,726       nogle af dine skud (CHR$
                                23). Der må kun være et
3E17            LD A,23         ad gangen. Hvis der er ét,
EDB1            CPIR            vil Z-flaget blive sat
E1              POP HL          hent HL ned – Z forbliver
                                uændret
```



```
C8              RET Z           Hvis der er et missil, hop til
                                CALL-stedet
CB65            BIT 4,L         Er nul nedtrykket?
C0              RET NZ          Hvis nej, hop også til CALL-
                                stedet
ED4BBC40        LD BC,(16572)   LOAD B med linienr., C ko-
                                lonnenr.
  (ED4BD543 – LD BC,(17365))
05              DEC B           Kuglen skal tegnes en linie
                                over
0C              INC C           kanonen. C incrementeres
                                to gange for
0C              INC C           at få skuddet lige over kano-
                                nen
CDF508          CALL 2293
  (CD940A – CALL 2708)
3E17            LD A,23
D7              RST 10h         Her tegnes et skud
C9              RET             Der returneres
```

Den næste rutine er MOVEMIS. Den flytter din bombe op mod invaderne. Den checker, om de rammer noget. Hvis de rammer dine skjold eller linien i linie 1, så skal bomben blot forsvinde. Hvis den rammer en af invader'nes bomber, skal begge forsvinde. Du skal vide, at vi har reserveret plads til antal points i DATA, og disse pladser er 16570 (17262) og 16571 (17364). Det første er at undersøge, om der i det hele taget er nogle af dine bomber på skærmen. Hvis der ikke er det, returneres der bare.

```
        MOVEMIS
2A0C40          LD HL,(16396)
  (217D40 – LD HL,16509)
01D602          LD BC,726
3E17            LD A,23
EDB1            CPIR
C0              RET NZ
```

Hvis der er nogle, fortsættes der, og den eftersøgte ligger på adresse (HL−1).

```
2B          DEC HL
```

På denne byte skal din bombe slettes.

Dine bomber går opad, og vi trækker derfor 33 fra HL, som er positionen. Vi skal nu se, hvad der står på den nye position.

```
3600        LD (HL),0
112100      LD DE,33
A7          AND A           Slet carry-flaget
ED52        SBC HL,DE
7E          LD A,(HL)
```

Nu skal vi finde ud af, om der skal laves en ny bombe, eller om der skal tælles point.

```
FE00        CP 0            Hvis pladsen ikke er fri,
2003        JR NZ,3         hoppes ned til CP 27
3617        LD (HL),23      ellers laves en bombe
C9          RET             og der returneres
FE1B        CP 27           Hvis pladsen ikke er en af
                            invadernes bomber, så
2003        JR NZ,3         hop til CP 3
3600        LD (HL),0       Ellers slettes deres bombe
C9          RET             og der returneres
FE03        CP 3            Hvis det er skjold eller top-
C8          RET Z           linie, skal der ikke laves en
                            ny bombe, og der returne-
                            res.
```

Nu har vi undersøgt en masse tegn. Hvis ingen af disse var det tegn, som vi har spurgt efter, er der kun en mulighed tilbage, og det er, at vi har ramt én af invaderne. Da vi ikke ved, om det er en CHR$ 6 eller CHR$ 134, der står på (HL), må vi decrementere (HL) og nu slette (HL), (HL+1) og (HL+2). Når vi gør det på

denne måde, kommer vi til at slette et mellemrum,men det gør jo ikke noget.

```
2B          DEC HL
0603        LD B,3
3600        LD (HL),0
23          INC HL
10FB        DJNZ −5
```

Så skal vi skrive AT 0,7, lige efter »SCORE=«.

```
010700      LD BC,7
CDF508      CALL 2293
  (CD940A − CALL 2708)
```

Her skal antal points stå. Antallet er lagret på 16570-71 (17363-64), men vi skal huske, at vi netop har ramt en invader, og da de giver et point pr. stk., skal vi forøge vor score med 1.

```
ED4BBA40    LD BC,(16570)
  (ED4BD343 − LD BC,(17363))
03          INC BC
ED43BA40    LD (16570),BC
  (ED43D343 − LD (17363),BC)
CD980A      CALL 2712
  (CD2D0C − CALL 3117)
```

På adresse 16576 (17369) har vi et tal, som holder styr på, hvor mange invadere, der er tilbage, og nu har vi skudt en ned.

```
21C040      LD HL,16576
  (21D943 − LD HL,17369)
35          DEC (HL)
C9          RET
```

Den næste rutine, MOVEBOMB, er den som flytter invadernes bomber, og dem kan der godt være flere end en af. Fremgangs-

måden er den samme som lige før. Først undersøges, om der er nogle. Deres bomber bevæges i øvrigt nedad.

```
MOVEBOMB
2A0C40          LD HL,(16396)
  (217D40 – LD HL,16509)
010003          LD BC,768
09              ADD HL,BC
3E1B    START   LD A,27         Deres bomber er et punk-
                                tum
EDB9            CPDR
C0              RET NZ
```

Læg mærke til, at vi undersøger skærmen bagfra.

Hvis der er nogle, har vi brug for at gemme HL og BC for at huske, hvor vi er nået til ved gennemscanningen. Vi vender jo tilbage for at se, om der er endnu flere. Hvor bomben befinder sig, kan vi se på (HL+1), da vi jo brugte CPDR.

```
C5              PUSH BC
E5              PUSH HL
23              INC HL
3600            LD,(HL),0       Slet bomben
```

Tegnet nedenunder – bomberne går ned – er på adresse (HL+33).

```
112100          LD DE,33
19              ADD HL,DE
7E              LD A,(HL)
```

Nu skal vi til at undersøge, hvad der evt. er ramt.

```
FE00            CP 0            Hvis det ikke er mellem-
2004            JR NZ,4         rum, hoppes til CP 23,
361B            LD (HL),27      ellers laves en ny bombe
1810            JR OM           og hop til OM
FE17            CP 23           Hvis det ikke er en af dine
```



```
2004            JR NZ,4         hoppes til CP 3,
3600            LD (HL),0       ellers slettes din bombe
1808            JR OM           og der hoppes til OM
FE03            CP 3            Hvis det er et skjold
2804            JR Z,OM         hoppes til OM
CBBF            RES 7,A
FE06            CP 6
E1      OM      POP HL
C1              POP BC
28D5            JR Z,START
```

I programmet er en RES 7,A-instruktion. En space invader er lavet af CHR$ 6 og CHR$ 134. Hvis bit 7 slettes, vil resultatet i begge tilfælde være 6. Dermed undersøges om invadernes egne bomber har ramt en af dem selv. Det må ikke give point, og bomben forsvinder. HL og BC POP'es tilbage fra stakken, og der vendes tilbage til START, hvis bomben ikke ramte din kanon. Det er det eneste, der ikke er undersøgt for. Hvis programmet fortsætter her, er du altså ramt.

Antal liv er lagret på adresse 16575 (17368), og det er altså blevet en mindre. Antallet af liv skal udskrives AT 0,29.

```
011D00          LD BC,29
CDF508          CALL 2293
  (CD940A – CALL 2708)
3ABF40          LD A,(16575)
  (3AD843 – LD A,(17368))
3D              DEC A
32BF40          LD (16575),A
  (32D843 – LD (17368),A)
4F              LD C,A
0600            LD B,0
CD980A          CALL 2712
  (CD2D0C – CALL 3117)
```

For at du kan se, at du er ramt, laver vi 3 lighedstegn oven på din kanon.

```
ED4BBC40        LD BC,(16572)
  (ED4BD543 – LD BC,(17365))
0C              INC C           1. tegn (mellemrum)
                                springes over
C5              PUSH BC
CDF508          CALL 2293
  (CD940A – CALL 2708)
3E14            LD A,20
D7              RST 10h
D7              RST 10h
D7              RST 10h
```

Så LOAD'er vi A med antallet af liv, og hvis det er nul, så er du helt færdig, og programmet dermed slut.

```
3ABF40          LD A,(16575)
  (3AD843 – LD A, (17368))
FE00            CP 0
200E            JR NZ,HER
```

Hvis du er død, skal der skrives »**SLUT**«, som starter på adresse 16553 (17346) i vor DATA-blok, og vi får endvidere rapportkoden 9 (ST) – STOP!

```
010B02          LD BC,523       AT 2,11
CDF508          CALL 2293
  (CD940A – CALL 2708)
21A940          LD HL,16553
  (21C243 – LD HL,17346)
CDC140          CALL PRINT (HL)
  (CDDA43 – CALL PRINT (HL))
CF              RST 8h
08              DEFB – 9 –
```

Hvis du har flere liv, skal lighedstegnene slettes, og vi holder en pause.

```
16FF    HER     LD D,255        DE er 256 * 255 + register E
CD3B42          CALL PAUSE
```



```
  (CD5445 – CALL PAUSE)
C1              POP BC          Du husker vel, at vi gemte
                                kanonpositionen
CDF508          CALL 2293
  (CD940A – CALL 2708)
3E00            LD A,0
D7              RST 10h
D7              RST 10h
D7              RST 10h
C34345          JP 17731
  (C35C48 – JP 18524)
1B      PAUSE   DEC DE
7B              LD A,E
B2              OR E
20FB            JR NZ,–5
C9              RET
```

Du undrer dig måske over JP 17731 (18524), men den hopper til et sted i hovedrutinen, som kører videre med spillet. PAUSE burde ikke volde problemer.

Næste rutine er den, som flytter invaderne, og det er den længste og sværeste underrutine. Vi kalder den MOVEINV.

Vi starter med at undersøge, om invaderne er på vej til venstre eller højre. Bit 0 i 16574 (17367) er 0, hvis de skal til højre, og 1, hvis de skal til venstre. Da HL skal LOAD'es med starten på udskriftfilen og BC med 726 i begge dele af rutinen, kan vi lige så godt gøre det her i starten.

```
MOVEINV
2A0C40          LD HL,(16396)
  (217D40 – LD HL,16509)
01D602          LD BC,726
3ABE40          LD A,(16574)
  (3AD743 – LD A,(17367))
CB47            BIT 0,A         Hvis bit 0 er 0,
283C            JR Z,HØJRE      så hop til HØJRE
```

Her må du passe på ikke at forveksle denne rutine kaldet HØJRE med rutinen af samme navn i KEYBSCAN.

```
3E06 START   LD A,6          Der undersøges om venstre
                             del er der
EDB1         CPIR
201D         JR NZ,CHECK     Hvis nej, hop til CHECK
C5           PUSH BC         Gem BC
E5           PUSH HL         og HL
2B           DEC HL          CHR$ står på (HL-1)
```

Vi skal også undersøge, om invaderen går ind i en af dine bomber. Adressen til venstre for CHR$ 6 er jo (HL-1).

```
2B           DEC HL
7E           LD A, (HL)
```

Hvis den går ind i en bombe, skal den forsvinde, og du får et point. Men vi har jo netop en rutine, som fjerner invaderen, og som også giver points. Det er allersidste del af den rutine, som flytter dine bomber. Så der skulle ikke være noget i vejen for at kalde sidste del af denne rutine – på adresse 16804 (17597). Summa summarum: Hvis der rammes, kaldes denne rutine, ellers flyttes invaderen til venstre.

```
FE17         CP 23           Hvis den har ramt en bom-
280D         JR Z,VPOINT     be, så hop til VPOINT.
54           LD D,H          Kopiér HL i DE.
5D           LD E,L
23           INC HL          Forøg HL.
010200       LD BC,2         2 tegn skal flyttes.
EDB0         LDIR            Flyt dem 1 til venstre. HL
                             er en større end DE.
2B           DEC HL
3600         LD (HL),0
1803         JR L3
VPOINT
CDA441       CALL 16804
```



```
  (CDBD44 – CALL 17597)
E1 L3        POP HL
C1           POP BC
18DD         JR START
```

Da vi flyttede invader'en én til venstre, skrev vi oven i venstre del af den invader, du flyttede. Højre halvdel skal slettes. HL peger nu på pladsen til højre for invaderen, så derfor står der DEC HL og LD (HL),0. Vi hopper over CALL-rutinen, for hvis der nås til JR 3, er invader'en ikke stødt ind i en bombe. Derefter POP'es HL og BC og næste invader findes.

Så kommer vi til CHECK. Denne del skal checke, om invaderen er nået helt ud til venstre side af skærmen. Det længste vi tillader dem at komme, er til kolonne 1. Så vi undersøger hele kolonnen for tegnet 6 (venstre halvdel af invaderen).

```
2A0C40 CHECK LD HL,(16396)   Identificer udskriftfilen
  (217D40 – LD HL,16509)
23           INC HL          Lad printposition pege på
                             kolonne nummer 1.
23           INC HL
112100       LD DE,33        Lad DE indeholde interval-
                             hop på 33.
0616         LD B,22         B indeholder antal linier.
3E06         LD A,6          Venstre invader-halvdel.
BE       CP  CP (HL)         Er der en invader?
2804         JR Z,BIT        Så hop til BIT.
19           ADD HL,DE       Ned til næste linie.
10FA         DJNZ CP
C9           RET             Returner.
```

Hvis de er nået helt til venstre skal bit 0 i 16574 (17367) resættes. Det betyder, at de skal gå til højre.

```
21BE40   BIT LD HL,16574
  (21D743 – LD HL,17367)
CB86         RES 0,(HL)
C9           RET
```

Så kommer vi til anden del af hele rutinen, og den flytter invaderne til højre. Fremgangsmåden er den samme som før. Denne gang undersøges skærmen bagfra. HL og BC kendes fra starten.

```
09      HØJRE   ADD HL,BC
3E86    START   LD A,134        Højre del af invaderen.
EDB9            CPDR            Hvis der ikke er flere,
201E            JR NZ,NED       hop til NED.
C5              PUSH BC
E5              PUSH HL
23              INC HL          Fremgangsmåden er som
                                før, men der decre-
23              INC HL          menteres.
7E              LD A,(HL)
FE17            CP 23
280D            JR Z,HPOINT
54              LD D,H
5D              LD E,L
2B              DEC HL
010200          LD BC,2
EDB8            LDDR
23              INC HL
3600            LD (HL),0
1804            JR 4
2B      HPOINT  DEC HL          Før invaderen slettes, skal
                                HL DEC'es to gange,
CDA341          CALL 16803      men på 16803 (17596)
  (CDBC44 – CALL 17596)         står netop DEC HL
E1              POP HL
C1              POP BC
18DC            JR START
```

Næste del af HØJRE er at undersøge, om invaderne er nået helt ud til højre på skærmen. Hvis de er det, skal de rykke en linie ned på skærmen mod dig.


```
011E02  NED     LD BC,542       Printpositionen
CDF508          CALL 2293       flyttes til AT 2,30.
  (CD940A – CALL 2708)
2A0E40          LD HL,(16398)   HL er lig printpositionen.
3E86            LD A,134
112100          LD DE,33
0616            LD B,22
BE      CP      CP (HL)
2804            JR Z,RYKNED
19              ADD HL,DE
10FA            DJNZ CP
C9              RET
```

Hvis de ikke er nået til højre, returneres der bare. Ellers skal bit 0 på 16574 (17367) sættes. Det indikerer jo, at de næste gang skal til venstre.

```
RYKNED
21BE40          LD HL,16574
  (21D743 – LD HL,17367)
CBC6            SET 0,(HL)
```

Nu skal vi til at flytte invaderne en linie ned. De skal findes en efter en og rykkes ned. Igen undersøger vi skærmen bagfra. Det gør vi af samme grund som de andre gange: Hvis man undersøger forfra – og man finder en – så bliver den jo flyttet en ned. Når vi så undersøger videre, finder vi jo den samme invader igen, og den skal da ikke flyttes en gang til . . .

```
2A0C40          LD HL,(16396)
  (217D40 – LD HL,16509)
01D602          LD BC,726       Hvis de allesammen
09              ADD HL,BC       er flyttet, så
3E86    START   LD A,134        hop til HELT-ND.
EDB9            CPDR
2026            JR NZ, HELT ND
SLET
```

```
C5          PUSH BC
E5          PUSH HL
3600        LD (HL),0       Der undersøges, om
23          INC HL          den er gået ned i et skud.
3600        LD (HL),0       Skuddet kan være både un-
                            der
012100      LD BC,33        CHR$ 6 og CHR$ 134, der
                            iøvrigt slettes.
09          ADD HL,BC       Hvis den er ramt,
7E          LD A,(HL)       skal den fjernes, og du får
FE17        CP 23           points. Men den er jo fjer-
2007        JR NZ,7         net i forvejen, og der
3600 SLET   LD (HL),0       skal dermed kun gives
                            points.
CDAB41      CALL 16811      Det starter ikke i 16803
                            (17596),
  (CDC444 – CALL 17604)     men 16811 (17604). 16803
                            (17596)
180B        JR OM           slettede jo først invaderen.
2B          DEC HL          Tegn invaderen igen
7E          LD A,(HL)
FE17        CP 23
28F3        JR Z,SLET
3606        LD (HL),6       Tegn invader'en igen
23          INC HL
3686        LD (HL),134
E1   OM     POP HL
C1          POP BC
18D4        JR START
```

Nu skal vi til at undersøge, hvor langt de er nået ned. Hvis de er nået til linie 21, er din base erobret, og du har dermed tabt.

```
010015 HELT ND
NEDE        LD BC,5376
CDF508      CALL 2293       Hele linie 21 undersøges,
  (CD940A – CALL 2708)
```



```
2A0E40      LD HL,(16398)   og hvis nogle er fundet,
012000      LD BC,32        bliver Z-flaget sat.
3E06        LD A,6
EDB1        CPIR
CA1C42      JP Z,16924      Er de det, så hop
  (CA3545 – JP Z,17717)
C9          RET             Returner
```

På adresse 16924 begynder den del, som skriver »**SLUT**« og stopper spillet.

Den sidste underrutine er den, som laver invadernes bomber. Den kalder vi BOMBE.

Fremgangsmåden er den, at der findes et tilfældigt tal mellem 0 og 31 (kolonnenummer). Derefter undersøges hele denne kolonne, og der startes nedefra. Hvis CHR$ 6 eller CHR$ 134 findes, skal et skud laves. Som du vel har opdaget, er invader'nes skud punktummer, CHR$ 27. Det er ikke sikkert, at den finder nogen i kolonnen, så vi laver en løkke, som gennemsøger flere kolonner.

```
BOMBE
060A        LD B,10         Antal gennemløb
C5   PUSH   PUSH BC         Gem BC
2A3240      LD HL,(16434)   Her laves det tilfældige tal.
54          LD D,H          Du kan se, at der hele tiden
5D          LD E,L          lægges tal til HL, og når
29          ADD HL,HL       HL er større end 65535, går
29          ADD HL,HL       HL jo bare videre forfra.
19          ADD HL,DE       16434 er system-variablen
29          ADD HL,HL       SEED, som styres af RAND.
29          ADD HL,HL       Grunden til, at der ikke bru-
29          ADD HL,HL       ges LD A, (16436) er, at ma-
19          ADD HL,DE       skinkode kører så hurtigt,
223240      LD (16434),HL   at variablen knapt nok
7C          LD A,H          ændrer sig. Og så er det jo
                            ikke ret tilfældigt. Tallet
                            lægges i A-registeret
```

```
E61F            AND 31          A bliver mellem 0 og 31
0614            LD B,20         Bomber må ikke placeres
4F              LD C,A          længere nede end 20
CDF508          CALL 2293
  (CD940A – CALL 2708)
```

Nu har vi i BC en position 20,x. Der skal undersøges i kolonne x. Systemvariabel 16398-99 indeholder adressen på den byte, der skrives på næste gang, så det benytter vi os af igen.

```
2A0E40          LD HL,(16398)   Denne løkke svarer til den
112100          LD DE,33        vi brugte, da vi undersøgte,
060F            LD B,15         om invaderne var kommet
7E      OM      LD A,(HL)       helt til højre
CBBF            RES 7,A
FE06            CP 6            Hvis en invader er fundet,
2809            JR Z,SKUD       hop til SKUD
A7              AND A
ED52            SBC HL,DE       Lad HL pege på linien
                                ovenover
10F4            DJNZ OM         Om igen
C1              POP BC          Skal flere kolonner under-
                                søges?
10D0            DJNZ PUSH       Hvis ja, hop til PUSH
C9              RET             Returner
19      SKUD    ADD HL,DE       Lad HL pege til adressen
                                under
361B            LD (HL),27      invader'ne, og lave en bom-
                                be.
C1              POP BC          Vi PUSH'ede i starten, og
C9              RET             den POP'er vi nu inden re-
                                turnering
```

Det var slut på den rutine. Nu resterer kun hoved-rutinen, men først skal du indtaste den følgende smule data. Disse data begynder iøvrigt på adresse 17226 (18019).


```
00808080800080808080008080808000 DATA DEFB SURPRISE
80808080008080808018008000000300
80000080008000008000800000800080
18008083838300800000800080000080
00800000000080838318000303038000
80808080008080808000800000000080
03031800830000800080000000008000
00800080000080008018008080808000
80000000008000008000808080800080
80808018188085048505800080858080
05808004858080058080808580800518
80858085058000000005850580008085
05000080008085051880850781058000
80858281058000808580800080808085
80800518808505800580008085078405
80008085050000808505000085051880
85050205028001850585058080018580
80058000808580800518 2E33382A3739
0028342E3318
```

Så kommer allersidste rutine, hovedrutinen, som begynder på adresse 17520 (18313). Det første er at bruge ovenstående data. Som du ser, er der mange divisionstegn (18h) imellem. Man kan næsten regne ud, at der vil komme til at stå noget på skærmen, men hvad det er, siger vi ikke.

```
214A43          LD HL,17226     Der bruges det samme,
  (216346 – LD HL,18019)
010304          LD BC,1027      som da vi lavede invaderne
04      NYLIN   INC B           i SKÆRM, bortset fra, at
78              LD A,B          vi går én linie ned ad
                                gangen.
FE11            CP 17           Vi har også brug for at
280C            JR Z,SILIN      huske, hvor vi er nået til
C5              PUSH BC         i DATA-blokken, derfor
E5              PUSH HL         PUSH'es HL hvis vi
CDF508          CALL 2293       når til linie 17
  (CD940A – CALL 2708)
```

```
E1              POP HL
CDC140          CALL PRINT (HL)
  (CDDA43 – CALL PRINT (HL))
C1              POP BC
18EE            JR NYLIN
04      SILIN   INC B           B bliver altså nu 18 (17+1)
0E0A            LD C,10         og C 10.
E5              PUSH HL         Gem HL.
CDF508          CALL 2293       Ny printposition: AT 18,10
  (CD940A – CALL 2708)
E1              POP HL          Hent HL tilbage.
CDC140          CALL PRINT (HL)Skriv sidste del af DATA.
  (CDDA43 – CALL PRINT (HL))
010015          LD BC,5376      Lav en streg i
CDF508          CALL 2293       linie 21.
  (CD940A – CALL 2708)
0620            LD B,32
CD2F41          CALL CHR3
  (CD4844 – CALL CHR3)
```

Nu er vores startbillede komplet. For at komme videre skal du trykke på BREAK.

```
CD460F          CALL 3910       Hvis BREAK ikke er aktive-
  (CD3B11 – CALL 4411)
38FB            JR C,–5         ret, så om igen.
CD2A0A          CALL 2602       Slet skærmen.
  (CDD00B – CALL 3024)
```

Dette var det indledende. Du husker vel følgende:

adresse 16570-71 (17363-64): antal points
16572-73 (17365-66): kanonens position
16574 (17367): afgør, hvilken vej invaderne rykker
16575 (17368): antal liv
16576 (17369): antal invaders på skærmen

AT 0,25 står der »LIV=«. Du skal jo vide, hvor mange du har i starten. Det skal stå AT 0,29.

```
011D00          LD BC,29        AT 0,29
CDF508          CALL 2293
  (CD940A – CALL 2708)
3E1F            LD A,31         Skriv tegnet »3«
D7              RST 10h         Du har altså 3 liv i starten.
212240          LD HL,16418     Vi bruger hele skærmen.
3600            LD (HL),0
210000          LD HL,0         Antal points skal
22BA40          LD (16570),HL   være lig nul, selvfølgelig.
  (22D343 – LD (17363),HL)
21BF40          LD HL,16575     Antal liv skal som sagt
  (21D843 – LD HL,17368)        være tre.
3603            LD (HL),3
```

Invader'ne skal starte med at gå til højre, dvs. bit 0 på 16574 (17367) skal være 0.

```
21BE40  LØK 1   LD HL,16574
  (21D743 – LD HL,17367)
CB86            RES 0,(HL)
210E16          LD HL,5646      Kanonens startposition skal
22BC40          LD (16572),HL   være AT 22,14.
  (22D543 – LD (17365),HL)
```

Vi har brug for en rutine, som sletter alle bomber på skærmen, når du dør eller har skudt alle ned. Den bliver lavet senere, men kaldt nu.

```
CD2745          CALL SLET
  (CD4048 – CALL SLET)
CDC940          CALL SKÆRM      Lav skærmen.
  (CDE243 – CALL SKÆRM)
21C040          LD HL,16576     Der er 40 invadere
  (21D943 – LD HL,17369)        i starten.
3628            LD (HL),40
```

Skærmen er nu blevet lavet, og alle invaders står der. Nu laver vi noget, som kun er til for sjov skyld. Vi vender skærmen 4 gange.

Denne rutine forklarer vi ikke, for det er den samme, som vi en gang gav et eksempel på.

```
0604            LD B,4          Antal vendinger.
C5      INV     PUSH BC         Gem BC.
1632            LD D,50
CD3B42          CALL PAUSE      En lille pause.
  (CD5445 – CALL PAUSE)
2A0C40          LD HL,(16396)
  (217D40 – LD HL,16509)
0E18            LD C,24
0620            LD B,32
23              INC HL
7E              LD A,(HL)
EE80            XOR 128
77              LD (HL),A
10F9            DJNZ –7
23              INC HL
0D              DEC C
20F3            JR NZ,–3
C1              POP BC
10E5            DJNZ INV
```

Næste del er den, som kalder alle vores underrutiner.

```
CD1243  LØK 2   CALL BOMBE
  (CD2B46 – CALL BOMBE)
CD7D41          CALL MOVEMIS
  (CD9644 – CALL MOVEMIS)
CDC241          CALL MOVEBOMB
  (CDDB44 – CALL MOVEBOMB)
CD3541          CALL KEYBSCAN
  (CD4E44 – CALL KEYBSCAN)
CD4142          CALL MOVEINV
  (CD5A45 – CALL MOVEINV)
```



```
CD7D41          CALL MOVEMIS
  (CD9644 – CALL MOVEMIS)
CDC241          CALL MOVEBOMB
  (CDDB44 – CALL MOVEBOMB)
CD3541          CALL KEYBSCAN
  (CD4E44 – CALL KEYBSCAN)
CD7D41          CALL MOVEMIS
  (CD9644 – CALL MOVEMIS)
CDC241          CALL MOVEBOMB
  (CDDB44 – CALL MOVEBOMB)
```

Du undrer dig måske over denne sammenhæng. Men din kanon skal bevæge sig hurtigere end invaderne. Rutinerne, som bevæger dine og deres bomber, må ikke være lagt lige efter hinanden. Tænk, hvis en bombe var 3 linier over dig. Så kunne du ikke nå at flytte dig.

```
3AC040          LD A,(16576)    A er antal invadere.
(3AD943 – LD A,(17369))
FE00            CP 0            Hvis der er flere,
20DB            JR NZ, LØK 2    så hop til LØK 2,
ED4BBC40        LD BC,(16572)   ellers
  (ED4BD543 – LD BC,(17365))
0C              INC C           slet din kanon
CDF508          CALL 2293       for at få en ny bane.
  (CD940A – CALL 2708)
3E00            LD A,0
D7              RST 10h
D7              RST 10h
D7              RST 10h
C3BF44          JP LØK 1        Hop til LØK 1.
  (C3D847 – JP LØK 1)
```

Nu kommer rutinen SLET, som før omtalt. Den er meget simpel.

```
2A0C40  SLET    LD HL,(16396)   HL er udskriftsfilens begyn-
                                delsesadresse
```

(217D40 – LD HL,16509)

01F702		LD BC,759	Undersøg 23 linier.
C5		PUSH BC	Gem BC.
3E1B		LD A,27	Invadernes bomber.
EDB1	L4	CPIR	Er der nogen?
2005		JR NZ,L5	Hvis nej, hop til L5
2B		DEC HL	Bomben står i (HL–1).
3600		LD (HL),0	Slet den.
18F7		JR L4	Hop til L4.
C1	L5	POP BC	Hent BC ned igen.
3E17		LD A,23	Dine bomber har kode 23
EDB9		CPDR	Er der en?
C0		RET NZ	Hvis nej, returner.
23		INC HL	Bomben står på (HL+1).
3600		LD (HL),0	Slet den.
C9		RET	Returner.

Du husker vel, at vi havde en JP-ordre i rutinen, som flytter invadernes bomber. JP hopper hertil, hvis du blev ramt. Da vi kaldte rutinen, blev en returadresse hængende på stakken. Væk med den.

E1 POP HL

Du skal nu have en ny position til kanonen, nemlig den, som man starter med – AT 20,14.

210E16	LD HL,5646	Ny position.
22BC40	LD (16572),HL	
(22D543 – LD (17365),HL)		
CD2745	CALL SLET	Slet bomber.
(CD4048 – CALL SLET)		
CD2141	CALL 16673	
(CD3A44 – CALL 17466)		

Med CALL 16673 (17466) kalder vi sidste del af SKÆRM-rutinen. Det er den, som laver din kanon. Og så den allersidste instruktion.

C3F244 JP LØK 2
(C30B48 – JP LØK 2)

Nu er programmet fuldstændigt. Inden du kører det med RAND USR 17520 (18313), som er starten af hovedrutinen, bør du gemme programmet på bånd. Hvis det crasher, kan du køre det tilbage og finde fejlen. Brug den lille rutine, som udskriver hekskoder, hvis du ikke har fået lavet en disassembler.

Hvis du synes, at bomberne går for hurtigt, kan du ændre i alle de CALL-ordrer, vi havde lige efter hinanden. Du kan jo også lave flere finesser i spillet, f.eks. højere score, jo højere oppe invaderne er, når de rammes. Du kan også lave det som på spillemaskinerne: Lade en UFO fare tværs over skærmen (i linie 2) en gang imellem, så du kan score ekstra-points med et velplaceret langskud.

Til Lambda-ejere kan vi sige, at dens indbyggede musik og lyd må gøre det muligt at finde en ROM-rutine, der kan sætte lidt gang i løjerne, når du bliver ramt, rammer invaderne osv.

Program 6

Dette program vil kunne gøre noget, som ZX 81 faktisk ikke er i stand til. Programmet vil gøre det muligt at spille musik på ZX 81, der tænker: »Hvis andre computere kan give koncert, så kan jeg da også. Hvorfor ikke?«

Programmet er kun lavet til ZX 81, da Lambda jo har indbygget musik, og herved forsvinder ideen bag programmet.

Programmet fylder såmænd ikke mere end 41 bytes, hvoraf 10 er DATA. Som sædvanlig skal du lave en REM-sætning på minimum 41 tegn, og sætningen skal placeres forrest i programmet. Via heks-loaderen skal du indtaste dette program:

Heks-kode	Mnemonic	Forklaring
9B897369 DATA	DEFB	Toner
53303538	DEFB	
3C46	DEFB	
CDBB02 START	CALL 699	Kald keyboardscanning.
CB44	BIT 0,H	Er SHIFT trykket ned,
C8	RET Z	så returner til BASIC.
44	LD B,H	Kopiér HL
4D	LD C,L	i BC.
2C	INC L	Hvis der ikke var aktiveret
28F5	JR Z, START	nogen tast, så hop til START.
CDBD07	CALL 1981	Hvilken tast aktiveret?
11F63F	LD DE,16374	Læg additionsfaktor i DE.
19	ADD HL,DE	Læg denne til tegn-adresse.
46	LD B,(HL)	Tag værdien i tabellen.
DBFF	IN A,(255)	Læg værdien på port 255 i A.
50	LD D,B	Kopier B i D.

10FE	L1	DJNZ L1	Tæl B ned til 0 (pause).
42		LD B,D	Hent B tilbage
D3FF		OUT (255),A	Læg A på port 255.
10FE	L2	DJNZ L2	Tæl B ned til 0 (pause).
18E1		JR START	Hop til start.

Keyboardscanningen skulle der ikke være meget besvær med. Hvis der ikke er aktiveret nogen taster, er register L LOAD'et med 255 (det er H i øvrigt også). Når vi så incrementerer L, vil L blive nul, hvis ingen taster er nedtrykket, og der hoppes til START for at finde ud af, om der er sket udvikling i sagen.

Det er meningen, at tasterne 1 til 0, altså øverste række, skal bruges til at spille på. Jo længere tasten sidder til højre, desto lysere (højere) bliver tonen. Når man bruger rutinen fra adresse 1981 i ROM, finder man jo ud af, hvilken tast der er trykket ned, og som sagt ligger disse i ROM fra adresse 126. Fra 140 til 149 ligger 1, 2, 3, 4, 5, 0, 9, 8, 7 og 6. Hvis vi trykker 1, vil HL vende tilbage med 140. Til dette tal lægges 16374, og i vores tilfælde vil HL nu indeholde 16514. Adresse 16514 er et sted i vores DATA; faktisk første byte. Denne bliver LOAD'et ind i register B. Som du kan se på DATA-listen, vil værdierne blive mindre og mindre, jo længere til højre du kommer (jo lysere tonen bliver). IN og OUT sørger for, at en tone bringes ud gennem fjernsynets højttaler. Mellem IN og OUT har vi tre instruktioner. Vi gemmer B i D, og tæller B ned til nul. Dette vil vi gøre, for at der kan komme en lille pause i programmet. B hentes tilbage igen, for vi skal bruge værdien til OUT. Efter denne instruktion kommer der endnu en pause, og der startes forfra.

Når du skal køre programmet, skal ZX 81 være i FAST-tilstand. Ellers virker programmet overhovedet ikke. Der startes med RAND USR 16524 – hvor CALL 699 begynder. Når du kører programmet, skal du selvfølgelig skrue op for lyden, og du skal muligvis også justere lidt på kanalvælgeren, selv om din nuværende indstilling giver det bedste billede. Det hele afhænger af, hvilket fjernsyn du har. Hvis kvaliteten er i orden, skulle du nu kunne spille melodier på ZX 81 – måske med lidt støj fra dit TV.

Du kan også lave programmet om, så alle tasterne kan bruges.

Til dette skal du udvide DATA en smule. De nye værdier kan du formindske eller forhøje lige meget i forhold til de oprindelige værdier. Så kan du også opnå at spille melodier i flere oktaver. Husk på, når du laver en ny DATA-liste, at rækkefølgen er korrekt. I kapitlet om keyboardscanningen har vi skrevet op, hvilken rækkefølge tegnene ligger i. Listen starter i 126. Du skal også være opmærksom på LD DE,16374. Det skal sikkert ændres. Hvis du får problemer med tonen, så prøv at se i ROM (med PEEK) hvilken adresse tegnet står i, og læg derefter DE til. Prøv så at se i DATA, om det er den rigtige værdi, der står der. Det vil det ikke være, hvis tonen er forkert.

Der er selvfølgelig også mulighed for at lave en DATA-liste i programmet. Musikprogrammet skal så selv aflæse fra denne liste, så det selv kan spille en melodi.

Hvis du har den første version, så prøv at spille den melodi, der hedder »Oh Susanna«. Opskriften følger her. Det kræver selvfølgelig, at du lever dig ind i melodien. Nogle af tasterne skal holdes nede lidt længere end andre. Det kommer helt af sig selv, hvis du kender lidt til melodien i forvejen. Her er de taster, du skal trykke på:

1 2 3 5 5 6 5 3 1 1 2 3 3 2 1 2
1 2 3 5 5 6 5 3 1 1 2 3 3 2 2 1
1 4 4 6 6 6 5 5 3 1 2
1 2 3 5 5 6 5 3 1 1 2 3 3 2 2 1

Appendiks A

I dette appendiks kan du finde en fuldstændig liste over heks-koder og deres tilhørende mnemonics.

n angiver et 8 bits-tal
nn angiver et 16 bits-tal
d angiver et tal mellem –128 og 127 (2-komplement-metoden)

Normale instruktioner

	0	1	2	3
0	NOP	LD BC,nn	LD (BC),A	INC BC
1	DJNZ d	LD DE,nn	LD (DE),A	INC DE
2	JR NZ,d	LD HL,nn	LD(nn),HL	INC HL
3	JR NC,d	LD SP,nn	LD(nn),A	INC SP
4	LD B,B	LD B,C	LD B,D	LD B,E
5	LD D,B	LD D,C	LD D,D	LD D,E
6	LD H,B	LD H,C	LD H,D	LD H,E
7	LD (HL),B	LD(HL),C	LD(HL),D	LD(HL),E
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E
9	SUB B	SUB C	SUB D	SUB E
A	AND B	AND C	AND D	AND E
B	OR B	OR C	OR D	OR E
C	RET NZ	POP BC	JP NZ,nn	JP nn
D	RET NC	POP DE	JP NC,nn	OUT (n),A
E	RET PO	POP HL	JP PO,nn	EX (SP),HL
F	RET P	POP AF	JP P,nn	DI

	4	5	6	7
0	INC B	DEC B	LD B,n	RLCA
1	INC D	DEC D	LD D,n	RLA
2	INC H	DEC H	LD H,n	DAA
3	INC (HL)	DEC (HL)	LD (HL),n	SCF
4	LD B,H	LD B,L	LD B,(HL)	LD B,A
5	LD D,H	LD D,L	LD D,(HL)	LD D,A
6	LD H,H	LD H,L	LD H,(HL)	LD H,A
7	LD (HL),H	LD (HL),L	HALT	LD (HL),A
8	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A
9	SUB H	SUB L	SUB (HL)	SUB A
A	AND H	AND L	AND (HL)	AND A
B	OR H	OR L	OR (HL)	OR A
C	CALL NZ,nn	PUSH BC	ADD A,n	RST 00h
D	CALL NC,nn	PUSH DE	SUB n	RST 10h
E	CALL PO,nn	PUSH HL	AND n	RST 20h
F	CALL P,nn	PUSH AF	OR n	RST 30h

	8	9	A	B
0	EX AF,AF'	ADD HL,BC	LD A,(BC)	DEC BC
1	JR d	ADD HL,DE	LD A,(DE)	DEC DE
2	JR Z,d	ADD HL,HL	LD HL,(nn)	DEC HL
3	JR C,d	ADD HL,SP	LD A,(nn)	DEC SP
4	LD C,B	LD C,C	LD C,D	LD C,E
5	LD E,B	LD E,C	LD E,D	LD E,E
6	LD L,B	LD L,C	LD L,D	LD L,E
7	LD A,B	LD A,C	LD A,D	LD A,E
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E
A	XOR B	XOR C	XOR D	XOR E
B	CP B	CP C	CP D	CP E
C	RET Z	RET	JP Z,nn	●
D	RET C	EXX	JP C,nn	IN A,(n)
E	RET PE	JP (HL)	JP PE,nn	EX DE,HL
F	RET M	LD SP,HL	JP M,nn	EI

	C	D	E	F
0	INC C	DEC C	LD C,n	RRCA
1	INC E	DEC E	LD E,n	RRA
2	INC L	DEC L	LD L,n	CPL
3	INC A	DEC A	LD A,n	CCF
4	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR H	XOR L	XOR(HL)	XOR A
B	CP H	CP L	CP(HL)	CP A
C	CALL Z,nn	CALL nn	ADC A,n	RST 08h
D	CALL C,nn	●	SBC A,n	RST 18h
E	CALL PE,nn	●	XOR n	RST 28h
F	CALL M,nn	●	CP n	RST 38h

Efter CB:

	0	1	2	3	4	5	6	7
0	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA(HL)	SLA A
3	–	–	–	–	–	–	–	–
4	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A
8	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6 (HL)	RES 6,A
C	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A

	8	9	A	B	C	D	E	F
0	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

Efter DD:

	0	1	2	3	4	5
0	–	–	–	–	–	–
1	–	–	–	–	–	–
2	–	LD IX,nn	LD (nn),IX	INC IX	–	–
3	–	–	–	–	INC (IX+d)	DEC (IX+d)
4	–	–	–	–	–	–
5	–	–	–	–	–	–
6	–	–	–	–	–	–
7	LD (IX+d),B	LD (IX+d),C	LD (IX+d),D	LD (IX+d),E	LD (IX+d),H	LD (IX+d),L
8	–	–	–	–	–	–
9	–	–	–	–	–	–
A	–	–	–	–	–	–
B	–	–	–	–	–	–
C	–	–	–	–	–	–
D	–	–	–	–	–	–
E	–	POP IX	–	EX (SP),IX	–	PUSH IX
F	–	–	–	–	–	–

	6	7	8	9	A	B	C	D	E	F
0	–	–	–	ADD IX,BC	–	–	–	–	–	–
1	–	–	–	ADD IX,DE	–	–	–	–	–	
2	–	–	–	ADD IX,IX	LD IX,(nn)	DEC IX	–	–	–	–
3	LD (IX+d),n	–	–	ADD IX,SP	–	–	–	–	–	–
4	LD B,(IX+d)	–	–	–	–	–	–	–	LD C,(IX+d)	–
5	LD D,(IX+d)	–	–	–	–	–	–	–	LD E,(IX+d)	–
6	LD H,(IX+d)	–	–	–	–	–	–	–	LD L,(IX+d)	–
7	–	LD (IX+d),A	–	–	–	–	–	–	LD A,(IX+d)	–
8	ADD A,(IX+d)	–	–	–	–	–	–	–	ADC A,(IX+d)	–
9	SUB (IX+d)	–	–	–	–	–	–	–	SBC (IX+d)	–
A	AND (IX+d)	–	–	–	–	–	–	–	XOR (IX+d)	–
B	OR (IX+d)	–	–	–	–	–	–	–	CP (IX+d)	–
C	–	–	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–	–	–
E	–	–	–	JP (IX)	–	EX DE,IX	–	–	–	–
F	–	–	–	LD SP,IX	–	–	–	–	–	–

Efter ED:

	0	1	2	3	4	5	6	7
0	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–
4	IN B,(C)	OUT(C),B	SBC HL,BC	LD (nn),BC	NEG	RETN	IM 0	LD I,A
5	IN D,(C)	OUT(C),D	SBC HL,DE	LD (nn),DE	–	–	IM 1	LD A,I
6	IN H,(C)	OUT(C),H	SBC HL,HL	–	–	–	–	RRD
7	–	–	SBC HL,SP	–	–	–	–	–
8	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–
A	LDI	CPI	INI	OUTI	–	–	–	–
B	LDIR	CPIR	INIR	OTIR	–	–	–	–
C	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–
E	–	–	–	–	–	–	–	–
F	–	–	–	–	–	–	–	–

	8	9	A	B	C	D	E	F
0	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–
4	IN C,(C)	OUT(C),C	ADC HL,BC	LD BC,(nn)	–	RETI	–	LD R,A
5	IN E,(C)	OUT(C),E	ADC HL,DE	LD DE,(nn)	–	–	IM 2	LD A,R
6	IN L,(C)	OUT(C),L	ADC HL,HL	–	–	–	–	RLD
7	IN A,(C)	OUT(C),A	ADC HL,SP	LD SP,(nn)	–	–	–	–
8	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–
A	LDD	CPD	IND	OUTD	–	–	–	–
B	LDDR	CPDR	INDR	OTDR	–	–	–	–
C	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–
E	–	–	–	–	–	–	–	–
F	–	–	–	–	–	–	–	–

Efter FD:

	0	1	2	3	4	5
0	–	–	–	–	–	–
1	–	–	–	–	–	–
2	–	LD IY,nn	LD(nn),IY	INC IY	–	–
3	–	–	–	–	INC (IY+d)	DEC (IY+d)
4	–	–	–	–	–	–
5	–	–	–	–	–	–
6	–	–	–	–	–	–
7	LD (IY+d),B	LD (IY+d),C	LD (IY+d),D	LD (IY+d),E	LD (IY+d),H	LD (IY+d),L
8	–	–	–	–	–	–
9	–	–	–	–	–	–
A	–	–	–	–	–	–
B	–	–	–	–	–	–
C	–	–	–	–	–	–
D	–	–	–	–	–	–
E	–	POP IY	–	EX (SP),IY	–	PUSH IY

	6	7	8	9	A	B	C	D	E	F
0	–	–	–	ADD IY,BC	–	–	–	–	–	–
1	–	–	–	ADD IY,DE	–	–	–	–	–	–
2	–	–	–	ADD IY,IY	LD IY,(nn)	DEC IY	–	–	–	–
3	LD (IY+d),n	–	–	ADD IY,SP	–	–	–	–	–	–
4	LD B,(IY+d)	–	–	–	–	–	–	–	LD C,(IY+d)	–
5	LD D,(IY+d)	–	–	–	–	–	–	–	LD E,(IY+d)	–
6	LD H,(IY+d)	–	–	–	–	–	–	–	LD L,(IY+d)	–
7	–	LD (IY+d),A	–	–	–	–	–	–	LD A,(IY+d)	–
8	ADD A,(IY+d)	–	–	–	–	–	–	–	ADC A,(IY+d)	–
9	SUB (IY+d)	–	–	–	–	–	–	–	SBC (IY+d)	–
A	AND (IY+d)	–	–	–	–	–	–	–	XOR (IY+d)	–
B	OR (IY+d)	–	–	–	–	–	–	–	CP (IY+d)	–
C	–	–	–	–	–	–	–	–	–	–
D	–	–	–	–	–	–	–	–	–	–
E	–	–	–	JP (IY)	–	EX DE,IY	–	–	–	–
F	–	–	–	LD SP,IY	–	–	–	–	–	–

Efter DDCBdd:

	6	E
0	RLC (IX+d)	RRC(IX+d)
1	RL (IX+d)	RR (IX+d)
2	SLA (IX+d)	SRA (IX+d)
3	–	SRL (IX+d)
4	BIT 0,(IX+d)	BIT 1,(IX+d)
5	BIT 2,(IX+d)	BIT 3,(IX+d)
6	BIT 4,(IX+d)	BIT 5,(IX+d)
7	BIT 6,(IX+d)	BIT 7,(IX+d)
8	RES 0,(IX+d)	RES 1,(IX+d)
9	RES 2,(IX+d)	RES 3,(IX+d)
A	RES 4,(IX+d)	RES 5,(IX+d)
B	RES 6,(IX+d)	RES 7,(IX+d)
C	SET 0,(IX+d)	SET 1,(IX+d)
D	SET 2,(IX+d)	SET 3,(IX+d)
E	SET 4,(IX+d)	SET 5,(IX+d)
F	SET 6,(IX+d)	SET 7,(IX+d)

Efter FDCBdd:

	6	E
0	RLC (IY+d)	RRC (IY+d)
1	RL (IY+d)	RR (IY+d)
2	SLA (IY+d)	SRA (IY+d)
3	–	SRL (IY+d)
4	BIT 0,(IY+d)	BIT 1,(IY+d)
5	BIT 2,(IY+d)	BIT 3,(IY+d)
6	BIT 4,(IY+d)	BIT 5,(IY+d)
7	BIT 6,(IY+d)	BIT 7,(IY+d)
8	RES 0,(IY+d)	RES 1,(IY+d)
9	RES 2,(IY+d)	RES 3,(IY+d)
A	RES 4,(IY+d)	RES 5,(IY+d)
B	RES 6,(IY+d)	RES 7,(IY+d)
C	SET 0,(IY+d)	SET 1,(IY+d)
D	SET 2,(IY+d)	SET 3,(IY+d)
E	SET 4,(IY+d)	SET 5,(IY+d)
F	SET 6,(IY+d)	SET 7,(IY+d)

Appendiks B

I dette appendiks finder du en liste over ZX 81's systemvariable.

Decimal	Heks	Navn	Formål
16384	4000	ERR NR	Rapportkoden minus 1
16385	4001	FLAGS	Tegngivere
16386	4002	ERR SP	Adressen for GOSUB-stakkens start
16388	4004	RAMTOP	Adressen for første byte efter BASIC
16390	4006	MODE	Specificerer markøren
16391	4007	PPC	Nummeret på den linie, der udføres
16393	4009	VERSN	
16394	400A	E PPC	Nummeret på den aktuelle linie
16396	400C	D-FILE	Startadressen for skærmbilledet
16398	400E	DF CC	Adressen på den næste printposition
16400	4010	VARS	Startadressen på variabelfilen
16402	4012	DEST	Adressen på den variabel, der bruges
16404	4014	E-LINE	Startadressen for den linie, der redigeres
16406	4016	CHADD	Adressen på den næste tegn, der oversættes
16408	4018	X ATR	Adressen på tegnet før syntaksmarkøren
16410	401A	STKBOT	Startadressen for beregnerstakken
16412	401C	STKEND	Slutadressen for beregnerstakken
16414	401E	BERG	Bruges ved beregninger
16415	401F	MEM	Startadressen for beregnerens RAM-område
16417	4021	SPARE1	Benyttes ikke
16418	4022	DF-SZ	Antallet af linier i nederste del af skærmen

16419	4023	S TOP	Linienummeret på øverste linie i automatisk listning
16421	4025	LAST K	Hvilken tast, der sidst er påvirket
16423	4027	DB ST	Tastaturets status
16424	4028	MARGIN	Antal linier over eller under billedet
16425	4029	NXTLIN	Startadresse på næste linie, der skal udføres
16427	402B	OLDPPC	Linienummer, som CONT hopper til
16429	402D	FLAGX	Tegngivere
16430	402E	STRLEN	Længden af streng under bearbejdelse
16432	4030	T ADDR	Adressen på næste enhed i syntakstabellen
16434	4032	SEED	Styrer RND
16436	4034	FRAMES	Holder styr på, hvor mange billeder der er vist
16438	4036	COORDS	Koordinaten for det sidst plottede punkt
16440	4038	PR CC	
16441	4039	S POSN	Koordinater for printposition
16443	403B	CD FLAG	Angiver, om ZX 81 er i SLOW eller FAST
16444	403C	PRBUFF	Fortæller printer, hvad der skal udskrives
16477	405D	MEMBOT	Bruges til beregninger, se kap. 22
16507	407B	SPARE2	Benyttes ikke

Appendiks C

Konvertering mellem talsystemer

DEC.	HEX.	DEC.	HEX.	DEC.	HEX.	2'sC.	DEC.	HEX.	2'sC.
0	00	64	40	128	80	−128	192	C0	−64
1	01	65	41	129	81	−127	193	C1	−63
2	02	66	42	130	82	−126	194	C2	−62
3	03	67	43	131	83	−125	195	C3	−61
4	04	68	44	132	84	−124	196	C4	−60
5	05	69	45	133	85	−123	197	C5	−59
6	06	70	46	134	86	−122	198	C6	−58
7	07	71	47	135	87	−121	199	C7	−57
8	08	72	48	136	88	−120	200	C8	−56
9	09	73	49	137	89	−119	201	C9	−55
10	0A	74	4A	138	8A	−118	202	CA	−54
11	0B	75	4B	139	8B	−117	203	CB	−53
12	0C	76	4C	140	8C	−116	204	CC	−52
13	0D	77	4D	141	8D	−115	205	CD	−51
14	0E	78	4E	142	8E	−114	206	CE	−50
15	0F	79	4F	143	8F	−113	207	CF	−49
16	10	80	50	144	90	−112	208	D0	−48
17	11	81	51	145	91	−111	209	D1	−47
18	12	82	52	146	92	−110	210	D2	−46
19	13	83	53	147	93	−109	211	D3	−45
20	14	84	54	148	94	−108	212	D4	−44
21	15	85	55	149	95	−107	213	D5	−43
22	16	86	56	150	96	−106	214	D6	−42
23	17	87	57	151	97	−105	215	D7	−41
24	18	88	58	152	98	−104	216	D8	−40
25	19	89	59	153	99	−103	217	D9	−39
26	1A	90	5A	154	9A	−102	218	DA	−38

27	1B		91	5B		155	9B	−101	219	DB	−37
28	1C		92	5C		156	9C	−100	220	DC	−36
29	1D		93	5D		157	9D	− 99	221	DD	−35
30	1E		94	5E		158	9E	− 98	222	DE	−34
31	1F		95	5F		159	9F	− 97	223	DF	−33
32	20		96	60		160	A0	− 96	224	E0	−32
33	21		97	61		161	A1	− 95	225	E1	−31
34	22		98	62		162	A2	− 94	226	E2	−30
35	23		99	63		163	A3	− 93	227	E3	−29
36	24		100	64		164	A4	− 92	228	E4	−28
37	25		101	65		165	A5	− 91	229	E5	−27
38	26		102	66		166	A6	− 90	230	E6	−26
39	27		103	67		167	A7	− 89	231	E7	−25
40	28		104	68		168	A8	− 88	232	E8	−24
41	29		105	69		169	A9	− 87	233	E9	−23
42	2A		106	6A		170	AA	− 86	234	EA	−22
43	2B		107	6B		171	AB	− 85	235	EB	−21
44	2C		108	6C		172	AC	− 84	236	EC	−20
45	2D		109	6D		173	AD	− 83	237	ED	−19
46	2E		110	6E		174	AE	− 82	238	EE	−18
47	2F		111	6F		175	AF	− 81	239	EF	−17
48	30		112	70		176	B0	− 80	240	F0	−16
49	31		113	71		177	B1	− 79	241	F1	−15
50	32		114	72		178	B2	− 78	242	F2	−14
51	33		115	73		179	B3	− 77	243	F3	−13
52	34		116	74		180	B4	− 76	244	F4	−12
53	35		117	75		181	B5	− 75	245	F5	−11
54	36		118	76		182	B6	− 74	246	F6	−10
55	37		119	77		183	B7	− 73	247	F7	− 9
56	38		120	78		184	B8	− 72	248	F8	− 8
57	39		121	79		185	B9	− 71	249	F9	− 7
58	3A		122	7A		186	BA	− 70	250	FA	− 6
59	3B		123	7B		187	BB	− 69	251	FB	− 5
60	3C		124	7C		188	BC	− 68	252	FC	− 4
61	3D		125	7D		189	BD	− 67	253	FD	− 3
62	3E		126	7E		190	BE	− 66	254	FE	− 2
63	3F		127	7F		191	BF	− 65	255	FF	− 1

Appendiks D

I dette appendiks kigger vi på flagsætning. Vi bruger følgende notationer:

1:	flaget sættes
0:	flaget resættes
*:	flaget uberørt
↓:	flaget ændres afhængigt af resultatet
P:	P/V-flaget viser partiet
V:	P/V-flaget indikerer overflow
X:	flagets værdi ukendt
S:	S-flaget (Sign-flaget)
Z:	Z-flaget (Zero-flaget)
P/V:	P/V-flaget (Partiet/Overflow-flaget)
C:	C-flaget (Carry-flaget)

	S	Z	P/V	C
8 bits-ADC	↓	↓	V	↓
8 bits-ADD	↓	↓	V	↓
16 bits-ADD	*	*	*	↓
AND	↓	↓	P	0
BIT	X	↓	X	*
CALL	*	*	*	*
CCF	*	*	*	↓
CP	↓	↓	V	↓
CPD	↓	↓	↓	*
CPDR	↓	↓	↓	*
CPI	↓	↓	↓	*
CPIR	↓	↓	↓	*
CPL	*	*	*	*
DAA	↓	↓	P	↓
8 bits-DEC	↓	↓	V	*
16 bits-DEC	*	*	*	*
DI	*	*	*	*
DJNZ	*	*	*	*
EI	*	*	*	*
EX	*	*	*	*
EXX	*	*	*	*
HALT	*	*	*	*
IM	*	*	*	*
IN A,n	*	*	*	*
IN r,(C)	↓	↓	P	*
8 bits-INC	↓	↓	V	*
16 bits-INC	*	*	*	*
IND	X	↓	X	*
INDR	X	1	X	*
INI	X	↓	X	*
INIR	X	1	X	*

	S	Z	P/V	C
JP	*	*	*	*
JR	*	*	*	*
LD_1	*	*	*	*
LDD	*	*	↓	*
LDDR	*	*	0	*
LDI	*	*	↓	*
LDIR	*	*	0	*
NEG	↓	↓	V	↓
NOP	*	*	*	*
OR	↓	↓	P	0
OUTI	X	↓	X	*
OTIR	X	1	X	*
OUT	*	*	*	*
OUTD	X	↓	X	*
OTDR	X	1	X	*
POP_2	*	*	*	*
PUSH	*	*	*	*
RES	*	*	*	*
RET	*	*	*	*
RETI	*	*	*	*
RETN	*	*	*	*
RL	↓	↓	P	↓
RLA	*	*	*	↓
RLC	↓	↓	P	↓
RLCA	*	*	*	↓
RLD	↓	↓	P	*
RR	↓	↓	P	↓
RRA	*	*	*	↓
RRC	↓	↓	P	↓
RRCA	*	*	*	↓
RRD	↓	↓	P	*
RST	*	*	*	*
8-bits SBC	↓	↓	V	↓
16-bits SBC	↓	↓	V	↓
SCF	*	*	*	1
SET	*	*	*	*
SLA	↓	↓	P	↓

	S	Z	P/V	C
SRA	↓	↓	P	↓
SRL	↓	↓	P	↓
SUB	↓	↓	V	↓
XOR	↓	↓	P	0

1) LD A,I og LD A,R påvirker S og Z afhængig af resultatet.
2) POP AF vil selvfølgelig ændre flagene afhængig af tallet på stakken.

Stikordsregister

ABS 160
ACS 160
Addition 36, 78, 154
Adresse 14
Akkumulator 22, 73, 150
Alternativt register 117
AND 49, 160
Andengradsligning 167
Antiscroll 99, 101
ASN 160
Assembler 11, 21
ATN 160

BASIC 11, 59, 90
BASIC-fortolker 11, 13
BCD-aritmetik 118
Beregnerstak 151
Bit 13, 19, 105
BIT 70
Blokinstruktioner 95, 102, 124, 125
BREAK 81, 84, 114
Byte 13, 19

CALL 80, 89
Carry-flag 33, 56, 63, 150
CCF 69
CHR$ 160
CLS 16
CODE 160
COS 160
CP 63
CPL 69
CPU 13

DAA 33, 118
DEFB 90
DI 125
Disassembler 140
Division 153
DJNZ 57

EI 125
Etiket 82
EX 64, 78, 117
EXP 160
EXX 117

FAST 140
Fejlrapportering 90
Flagregister 32, 33, 73
FRAMES 93

GOSUB 80

HALT 125
Heks-kode 21, 121
Heks-loader 21
Hop 56

IN 123
Indeksregister 45, 111
INT 160
Interrupt 125
Interrupt-register 46
Inverterings rutine 58-59

JP 56, 60
JR 56

Keyboard 131
Kurve 175

Kvadratrod 152

LEN 160
LN 160
Load 22, 25, 28, 30, 31, 32, 140
Logiske operatorer 49
LSB 15, 64

Matematik 151
MEMBOT 163, 168
Mnemonic 11, 21
MSB 15, 24, 26, 64
Multiplikation 151
Musik 213, 214
Mønstre 87

NEG 69
NEW 89
Newline 64, 83, 125
NOP 69, 125
NOT 160

Ombytning 64, 78, 117, 157
OR 51, 160
OUT 124
Overflowindikator 43

Paritetsindikator 33, 51
Parity/overflow-flag 33
Pause 60
PEEK 14, 15, 160, 162
PLOT 138
POKE 15, 29
POP 74, 85
Port 123, 124
Potensfunktion 160, 162
PRINT AT 137
Printposition 16
Program Counter 56
Programfil 14, 15, 64
Programpil 56
PUSH 72

RAM 13
RAMTOP 14, 16, 89, 148
RAND 24
Refresh-register 46
Register 24, 48, 117
REM 148, 164
Renumber 64, 68
RES 70
RET 24, 85, 125
ROM 13, 14, 136
Rotationsinstruktioner 105, 108, 110
RST 89, 118, 151
Rutine 24

Sammenligning 63
SAVE 140, 148
SCF 69
SCL 13
Scroll 97, 99, 139
SET 69
SGN 160
Sign bit 33, 42
Sign-flag 33
SIN 160, 175
Skifteinstruktioner 105, 110
Skrivemaskine 165
Skærmbillede 15, 16, 97
SLOW 140
Stackpointer 72
Stak 72
STR$ 160
Subrutine 80
Subtraktion 36, 78, 153
Systemvariable 14, 93, 229

Talsystem, binært 18
Talsystem, decimalt 18, 139, 231
Talsystem, heksadecimalt 18, 231
TAN 160
Tastatur 131
Tegn 71, 111, 116, 136
Tegngenerator 111
Tekstbehandling 165
Tokomplementmetode 42, 231
Trigonometriske funktioner 160, 175
Tvilling-register 117

Udskriftfil 14, 15, 27
Udvidelsesmodul 12
ULA 13
Underrutine 80

USR 24, 89, 160

VAL 160
Variabelfil 14, 17, 167
Vending 58-59

XOR 53

Zero-flag 33, 56, 63
Z80-processor 13

ZX81 er en lille uanseelig maskine og mange tvivler derfor i starten lidt på dens ydeevne. Man glemmer, at der inden i maskinen sidder en Z80 mikroprocessor mage til den, som sidder i mange professionelle datamater.

Ikke desto mindre er det rigtigt, at maskinen i visse tilfælde kan være lidt langsom, når den programmeres i BASIC.

Den fulde ydeevne får man derimod frem ved at bruge maskinkode. Maskinkode er sværere end BASIC, men rummer forbavsende muligheder for at styre maskinens præstationer. Faktisk kan visse maskinkodeprogrammer, når de udføres, virke næsten som trylleri.

Maskinkode med ZX81 er en bog for begyndere. Den er skrevet af to gymnasieelever, og alting forklares til mindste detalje, helt fra grunden.

Bogen kan også bruges sammen med den ZX81-lignende maskine Lambda.

Borgen

ISBN 87-418-7245-2